REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE 25.Oct.99	3. REPORT TYPE AND DATES COVERED THESIS

4. TITLE AND SUBTITLE MULTIRESOLUTION IMAGE FUSION OF THEMATIC MAPPER IMAGERY WITH SYNTHETIC APERTURE RADAR IMAGERY	5. FUNDING NUMBERS
6. AUTHOR(S) 2D LT MEEK THEODORE R	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) UTAH STATE UNIVERSITY	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) THE DEPARTMENT OF THE AIR FORCE AFIT/CIA, BLDG 125 2950 P STREET WPAFB OH 45433	10. SPONSORING/MONITORING AGENCY REPORT NUMBER FY99-339
11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION AVAILABILITY STATEMENT Unlimited distribution In Accordance With AFI 35-205/AFIT Sup 1	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

DISTRIBUTION STATEMENT A
Approved for Public Release
Distribution Unlimited

19991117 076

14. SUBJECT TERMS	15. NUMBER OF PAGES 114
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT

DTIC QUALITY INSPECTED 4

MULTIRESOLUTION IMAGE FUSION OF THEMATIC MAPPER IMAGERY WITH SYNTHETIC APERTURE RADAR IMAGERY

by

Theodore R. Meek

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Electrical Engineering

Approved:

[illegible]
Dr. Doran J. Baker
Major Professor

Cynthia Furse
Dr. Cynthia M. Furse
Committee Member

Scott E. Budge
Dr. Scott E. Budge
Committee Member

[illegible]
Dr. James P. Shaver
Dean of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

1999

ABSTRACT

Multiresolution Image Fusion of Thematic Mapper Imagery

with Synthetic Aperture Radar Imagery

by

Theodore R. Meek, Master of Science

Utah State University, 1999

Major Professor: Dr. Doran J. Baker
Department: Electrical and Computer Engineering

This study was designed to demonstrate the feasibility of applying multiresolution image fusion techniques to synthetic aperture radar (SAR) and Landsat imagery. This was accomplished through the development and application of image fusion software to test images, for the purpose of comparing results, to show that information from more than three bands can be used to study surface and subsurface features in a single image. The test images were fused using six image fusion techniques that are the combinations from three types of image decomposition algorithms (ratio of low pass [RoLP] pyramids, gradient pyramids, and morphological pyramids) and two types of fusion algorithms (selection and hybrid selection and averaging). Based upon the composite images formed by fusing the test images, this study concludes that: small details in city areas make morphological pyramids ineffective, selection forms of fusion do not effectively combine the data, RoLP and gradient pyramids with hybrid fusion produce the best results, and optimum pyramid depth is dependent upon the size of detail in the images. (122 pages)

ACKNOWLEDGMENTS

I would like to thank my committee members, Dr. Cynthia Furse and Dr. Scott Budge, for their help and support as well as for the time spent in service to me as my master's committee. Special thanks go to Dr. Doran Baker, my major professor, for his help, support, time, and resources. I greatly appreciate the work my committee has done on my behalf. I would also like to thank David Oliver and Bill Pfaff for their help in obtaining test images.

I would also like to thank the Rocky Mountain NASA Space Grant Consortium for the fellowship that allowed me to remain at Utah State University to earn a master's degree. The office staff, Kim, Lindsey, Jesalee, and Kristine, also deserve great thanks for all of the help they have given me. I would also like to thank the United States Air Force Institute of Technology for selecting me as an AFIT student and Lieutenant Colonel Johnny Hays and Major Victoria Rondeau (retired) for helping me remain at Utah State University.

A very special thank you also goes to my wife, Jessica, for supporting me even when times were hard and to our daughter, Sabrina, for providing me with the pleasant diversions that help keep a student motivated. The greatest thanks go to my God to whom I owe all of my success and rewards and knowledge.

Theodore R. Meek

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

A. *Remote Sensing and Earth Images*

Remote sensing is the technique of using sensors located a considerable distance from the information source to obtain, process, and interpret data to obtain information and perspective about the source itself. Terrestrial remote sensing has become popular in the last two decades because it makes possible easy mapping, monitoring, and study of changes on or even beneath the Earth's surface. Images obtained from remote sensors have been used for the study of ice sheets [1, 2], locating villages in third world countries [3], locating gold, vegetation mapping, land use studies, geological mapping, watershed studies, urban planning, and much more [4, 5].

A widely used source of remotely sensed data is the Landsat satellite series. Landsat uses the optical and near infrared (IR) wavelengths. Low spatial resolution (83 meters) multispectral scanner instrument images from the first Landsat satellite, Landsat-1, have been available since 1972 [6]. Although Landsat images are an excellent source of information, the images can be quite expensive. High spatial resolution (30 meters) images from Landsat-5's Thematic Mapper (TM) instrument can be obtained through Space Imaging EOSAT but cost thousands of dollars per scene. A less expensive source is the EROS Data Center that sells images over 10 years old for $200 to $400 per scene [6]. There have been five Landsat satellites successfully launched by the United States; Landsat-6 was lost, and Landsat-7 is scheduled for launch in April 1999.

Another source of remotely sensed data is synthetic aperture radar (SAR). SAR images of the Earth's surface have been available since the late 1970s. The first U.S. SAR satellite was Seasat, which was launched in 1978 to collect data on sea-surface winds, sea ice features, and ocean topography [4]. The Space Shuttle has flown three SAR instruments: SIR-A in 1982, SIR-B in 1985, and SIR-C in 1994. SAR has become increasingly popular in the 1990s, most likely because of new image processing techniques that incorporate accurate correction of acquisition errors such as foreshortening. Several countries have launched SAR satellites; these include the European Space Agency's ERS-1 and ERS-2 launched in 1991 and 1996, Japan's JERS-1 launched in 1992, and Canada's Radarsat launched in 1995 [4].

SAR has a distinct advantage over passive sensors, such as the ones in Landsat satellites, in that SAR does not rely on solar illumination. In fact, SAR can gather data in both sunlit and dark regions as well as in regions covered by clouds, fog, or pollution. SAR can even penetrate several near-surface materials such as sand, foliage, and forest canopies [4, 7]. This capability makes SAR an excellent companion for study with optical sensors.

B. Viewing Techniques and Challenges

Most remotely sensed data come in bands corresponding to the wavelength used to acquire the images. Typical Landsat TM images have seven bands, and SAR images can have several bands. In order to view a single band, there are two main options, representing the image as a grayscale image or as a pseudocolored image.

To view a single band using the grayscale option, each sensor sample, or pixel, will be assigned a value, usually between 0 and 255. If the sensor does not receive or store the samples on a 0 to 255 scale, the conversion is simple: The highest value in the range corresponds to 255; the lowest corresponds to 0, and all other values are interpolated and quantized to fit the 0 to 255 scale. Color pictures are made up of three grayscale images, a red image, a green image, and a blue image (RGB), commonly called RGB values. For example, the expression RGB = (1, 4, 2) would mean that the red pixels are band 1, green pixels are band 4, and blue pixels are band 2. In a grayscale image, it is often difficult to distinguish between pixels of similar value because they seem to blend together. Pseudocoloring is used because the human eye readily detects contrast in images [8, 9]. The pseudocoloring technique is to assign each pixel intensity a red, green, and blue corresponding color level. This scale helps viewers see large-scale changes across the image. The color level assigned to each pixel value can be arbitrary; however, the following is a common scale: Pixel intensity from 0 to 47 changes from black to violet; intensity from 48 to 79 changes from violet to blue; and additional increments of 47 go from blue to cyan to green to yellow to red to white [5]. White is still the most intense pixel, and black is the least intense.

To view multiple bands, several techniques have been implemented. Nonetheless, each technique is based upon the same principle: Assign one band to each pixel color red, green, and blue. This limits the viewing of an image to three bands; however, it allows full color images of the data. Typical Landsat images are viewed with color combinations that show the bands of interest to the study; different bands show different properties on the Earth's surface. More details on what specific Landsat bands show will be given in Chapter II. Other than a simple band being used to represent the color, ratios of bands have also been used. For example, red could be band 7 divided by band 4; green could be band 2 divided by band 1, and blue could be band 7 divided by band 3 [10]. SAR images with more than one band can also be viewed this way; for example, red could be band L, and green and blue could both be band C [5].

Composite images can be formed if proper image registration has been achieved. This usually requires extensive processing of the SAR images to correct for foreshortening. This is done by assigning bands from both sources to the red, green, and blue pixel values. An example would be to use two Landsat bands for red and green and a SAR band for blue. This technique has proven extremely useful in determining land usage; however, this technique is still limited because only three bands can be viewed simultaneously.

C. *Multiresolution Image Fusion*

Combining two images to form a composite image is not unique to remote sensing. Several applications have use for image fusion, i.e., the merging of two images to create one. Image fusion can be used for computer vision, aiding pilots to land in inclement weather, compressing source image data, medical diagnosis through fusing CT and MR images, ease of viewing to reduce human operator work load, etc. [11, 12, 13]

The simplest approach to image fusion is to set the composite pixel value equal to the average of the source values. This is not the optimal approach, however, because features that appear in one image and not the other are shown in the composite at a reduced contrast or are superimposed on features from the other image.

The key to successful image fusion is to create a composite image that retains all useful information contained in the source images without introducing artifacts that can interfere with analysis and interpretation [11, 12]. Figure 1 represents the approach taken in current literature, a multiresolution pyramid [11-17]. In this approach, the source images are filtered and reduced repetitively, forming a pyramid. In the figure, A_0 and B_0 are the original source images to be fused; A_1 and B_1 are reduced-

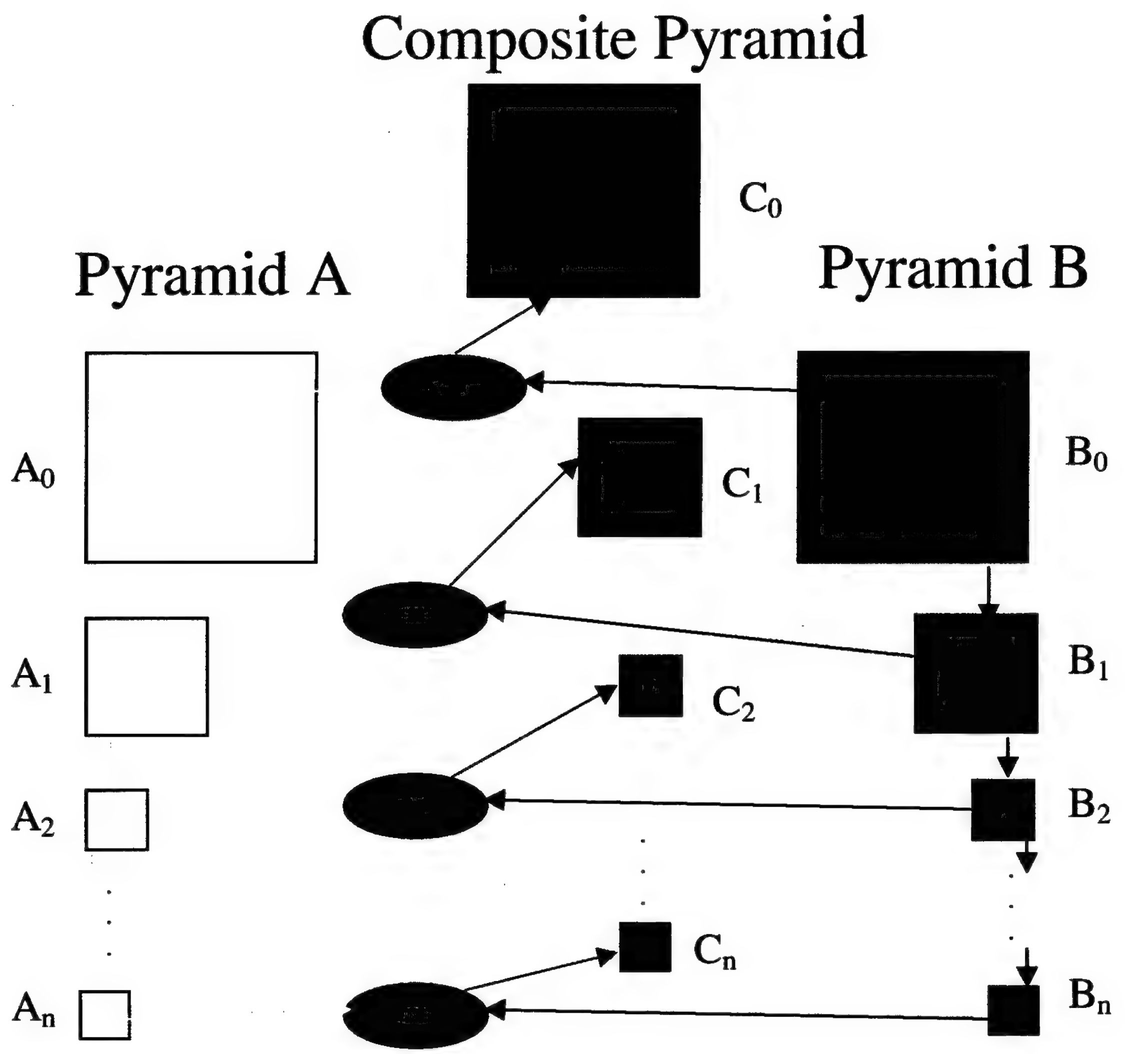


Fig. 1. Data flow diagram for pyramid fusion process

filtered versions of A_0 and B_0. Likewise, A_2 and B_2 are reduced-filtered images of A_1 and B_1. This process will be explained in more detail in Chapter III. Each level of the pyramid is then fused, using a fusion algorithm to create a composite pyramid. The fusion process can be either complex or simple. (Fusion processes are discussed in Chapter IV.) A composite image is generated from the composite pyramid, which will include all important information from the source images and no new artifacts. Figure 1 is a data flow diagram of the multiresolution fusion process. The square boxes represent the levels of the pyramid, or images. The red circles represent functions that fuse the two incoming images into one composite outgoing image. Although no function appears between levels of the pyramids, it should be understood that, when constructing a pyramid from the source image, there is a function that filters and subsamples the current image to generate the higher level. The green boxes represent the composite pyramid; however, C_0 is not the final composite image. The final image is obtained by reconstructing the composite pyramid.

D. Study Approach

Analysis of Landsat and SAR images is conventionally accomplished by assigning particular bands or ratios of bands to red, green, and blue pixels. By choosing specific bands or ratios of bands, analysis of particular properties can be made easier. In several cases it is possible to generate images from different bands that show the same properties. A popular way to analyze Landsat and SAR images of the same areas concurrently is done by overlaying bands from one source with bands of another source after the images have been properly registered [1, 2, 3, 5, 7], for example, having red and green pixels represent two Landsat bands and blue pixels represent a SAR band.

Current image fusion techniques can create a composite image that contains all of the important information in the source images without adding new artifacts to interfere with image analysis. Since a human observer cannot reliably integrate visual information by viewing multiple images separately and consecutively [17], image fusion makes it possible for the observer to view one image that contains all of the information contained in the source images. The present study includes programming the image fusion techniques described in current literature and applying them to Landsat and SAR images to diagnose the usability of each fusion technique as applied to remotely sensed imagery. The results and composite images are given in Chapter V.

E. Study Objectives

The first objective of this study was to compare current image fusion techniques and diagnose their effectiveness in fusing SAR and Landsat imagery based upon the composite image results obtained through the fusion of real SAR and Landsat images and the time required to generate the composite image. This was accomplished by applying the fusion techniques to several test images, recording the time required for each technique to produce the composite image, and visually inspecting the image to observe composite image quality.

The second objective was to combine test images by applying the fusion techniques to real SAR and Landsat image data to obtain useful information from more than three remote sensor bands. This was accomplished by creating test images designed to show specific surface features and then fusing the test images to verify that the surface features have been successfully fused.

The third objective of this study was to determine the optimum pyramid depth for the fusion of remotely sensed imagery. This was determined by examining the same set of test images fused over a range of pyramid depths.

CHAPTER II

LANDSAT AND SAR IMAGES

A. *Landsat Image Characteristics*

The Landsat images used in this study were acquired by Landsat-4's TM sensor recorded on 25 September 1994 in path number 38, row number 31. The images were purchased/requested by the Rocky Mountain NASA Space Grant Consortium (RMNSGC) and have been used in previous studies by both David Oliver in 1996 [5] and Bill Pfaff in 1997 [18].

Landsat TM images do not have significant geometric distortions because the viewing angle of the sensors is nearly perpendicular to the Earth's surface. In TM images, each pixel is stored as an 8-bit number and represents either a 30x30 meter or 120x120 meter terrestrial area. The Landsat satellite is in low Earth orbit with an altitude of 705 km and a 16 day, 233 orbit cycle. The TM sensor has seven bands; bandwidths, resolutions, and percents of transmission for the TM bands are listed in Table I. The percent of transmission for each band is the minimum value of transmission within the corresponding bandwidths. Values of transmission were taken from [19].

Each of the seven bands was designed to maximize detecting and monitoring of different types of Earth resources [20]. Band 1 is in the visible spectrum corresponding to blue light; it penetrates water for bathymetric mapping along coastal areas and is used for soil/vegetation differentiation and for distinguishing forest types. Band 2 is also in the visible spectral region corresponding to green light; it is used to detect green reflectance from healthy vegetation. Band 3 is in the visible spectrum corresponding to

TABLE I
LANDSAT TM BAND INFORMATION

Band	Bandwidth (μm)	Resolution (m)	% Transmission
1	0.45 – 0.52	30	100
2	0.53 – 0.61	30	95
3	0.62 – 0.69	30	100
4	0.78 – 0.91	30	84
5	1.57 – 1.78	30	98
6	10.42 – 11.66	120	88
7	2.08 – 2.35	30	98

red light; it is designed to detect chlorophyll absorption in vegetation. Band 4 is in the near-infrared and is used for detecting reflectance peaks in healthy green vegetation and for detecting water-land interfaces. Bands 5 and 7 respond in the near-infrared and are used for vegetation and soil moisture observations as well as for discriminating between rock and mineral types. Band 6 is selected for the atmospheric spectral window and is in the thermal-infrared spectrum, designed to assist in thermal mapping and in soil moisture and vegetation studies [5, 20].

B. SAR Image Characteristics

Synthetic aperture radar is based on technology that uses the motion of the sensor to synthesize an antenna aperture larger than the physical antenna [4]. This technique provides an enhanced spatial resolution imaging capability that is, to the first order, independent of sensor altitude. SAR functions by sending a microwave signal at the target and recording the returned signal. Being an active rather than a passive system, SAR has the advantage of being independent of solar illumination of the Earth's surface. SAR return signals are not inhibited when the Earth's surface is covered by clouds, fog, haze, or smoke; in fact, SAR can even penetrate the Earth's surface to return information about subsurface features [7] or can penetrate forest canopies that obscure the ground [4]. For these reasons, SAR can produce radar terrain maps that show the presence of major subjacent features not shown in Landsat images at the same spatial resolution [7].

SAR images are produced from the signals reflected from the Earth's surface. The strength of the return signal is dependent upon both the surface characteristics and the radar system. Table II gives a list of parameters that affect the return signal. When the surface roughness is at the same scale as the radar wavelength, signal interference occurs resulting in a speckled image. The look angle can also cause problems, such as geometric distortion and the appearance of near range pixels being brighter than far range pixels, in the image [1].

There are numerous possible SAR wavelength bands; however, unlike the Landsat TM imagery, different SAR bands are identified by letters. The surface properties proposed for study should be used to determine the band used; different bands show different properties. When viewing single-band SAR images, a user can make use

TABLE II
PARAMETERS AFFECTING SAR RETURN SIGNALS

Ground Parameters	Radar System Parameters
Terrain Slope and Texture	Frequency, Wavelength
Surface Roughness	Radar Look Angle/Depression Angle
Complex Dielectric Constant	Antenna Look Direction
Terrain Feature Orientation	Polarization
Shadowing	Signal-to-Noise Ratio

of the same grayscale or pseudocoloring techniques discussed for TM images in Chapter I. For SAR images, the highest pixel values represent areas of saturated radar return; these areas would appear white in the coloring schemes mentioned.

Images from the SIR-C SAR aboard STS-68 are used in this study. SIR-C operated at two frequencies, the C band at 5.3 GHz and the L band at 1.24 GHz. Each band has five images that correspond to different polarization modes. The five modes are designated as HH, HV, VH, VV, and total power. The first letter represents the transmitted signal, and the second letter represents the return signal; H designates horizontal polarization, and V stands for vertical polarization. Therefore, the VH image would be the data acquired from a vertically polarized transmitter and a horizontally polarized receiver. Total power images are the sum of the horizontal and vertical transmissions and the horizontal and vertical receptions. In this study, the L and C band total power images are used. These images, like the Landsat images, were acquired by the RMNSGC [5]. The SIR-C sensor look angle is 35 degrees off nadir. (Nadir is straight down.) Each pixel covers a 12.5 meter by 12.5 meter area.

C. Combined Image Characteristics

The capability of SAR to operate in either light or dark all-weather environments, coupled with the fact that radar terrain maps made from SAR data show the presence of major subjacent features that are not detectable in Landsat pictures at the same scale, implies that SAR would be an excellent complement to Landsat images. However, the look angle of the SAR sensor results in large geometric distortions in the resulting image display. Due to these large geometric distortions, some previously integrated SAR and

Landsat studies concluded that SAR added little value because of scene registration problems. New SAR image processing techniques have been used to overcome this problem, and SAR images can now be registered to Landsat images with errors reduced to the order of one pixel [5]. Integrated Landsat and SAR images have been used to aid in geologic mapping [10, 21], the study of ice sheets [1], and for land use studies [5]. One article suggested that neither the Landsat TM nor the SIR-B SAR imagery alone could distinguish the land use patterns in Sudan, while an integrated SAR and Landsat approach was successful [3].

In the present study only images that have already been processed and registered are used. Correction of geometric distortions of the SAR images and SAR and Landsat image registration was performed by David Oliver as part of his thesis in 1996 [5]. Figure 2 and Figure 3 are the Landsat-4 and SAR images of Cache Valley used in this study. Figure 2 is composed of Landsat-4 TM bands 1, 4, and 2 corresponding to the red, green, and blue pixel values, respectively. Figure 3 is composed of the SAR L and C total power bands. The red pixels correspond to the L band, and the green and blue pixels correspond to the C band. Note that in the upper right-hand corner of Figure 3, the SAR image contains no data; this is due to the correction for geometric distortion.

Corresponding subsections of the images in Figure 1 and in Figure 2 are fused, using the multiresolution techniques described in current literature. The usefulness of these fusion techniques as applied to remote sensed imagery will be assessed in Chapter V. Different band combinations can be used to show different surface/subsurface properties; in several cases, there are more than three bands that can be used to show the same features. To more effectively analyze multiple images, image fusion can be used.

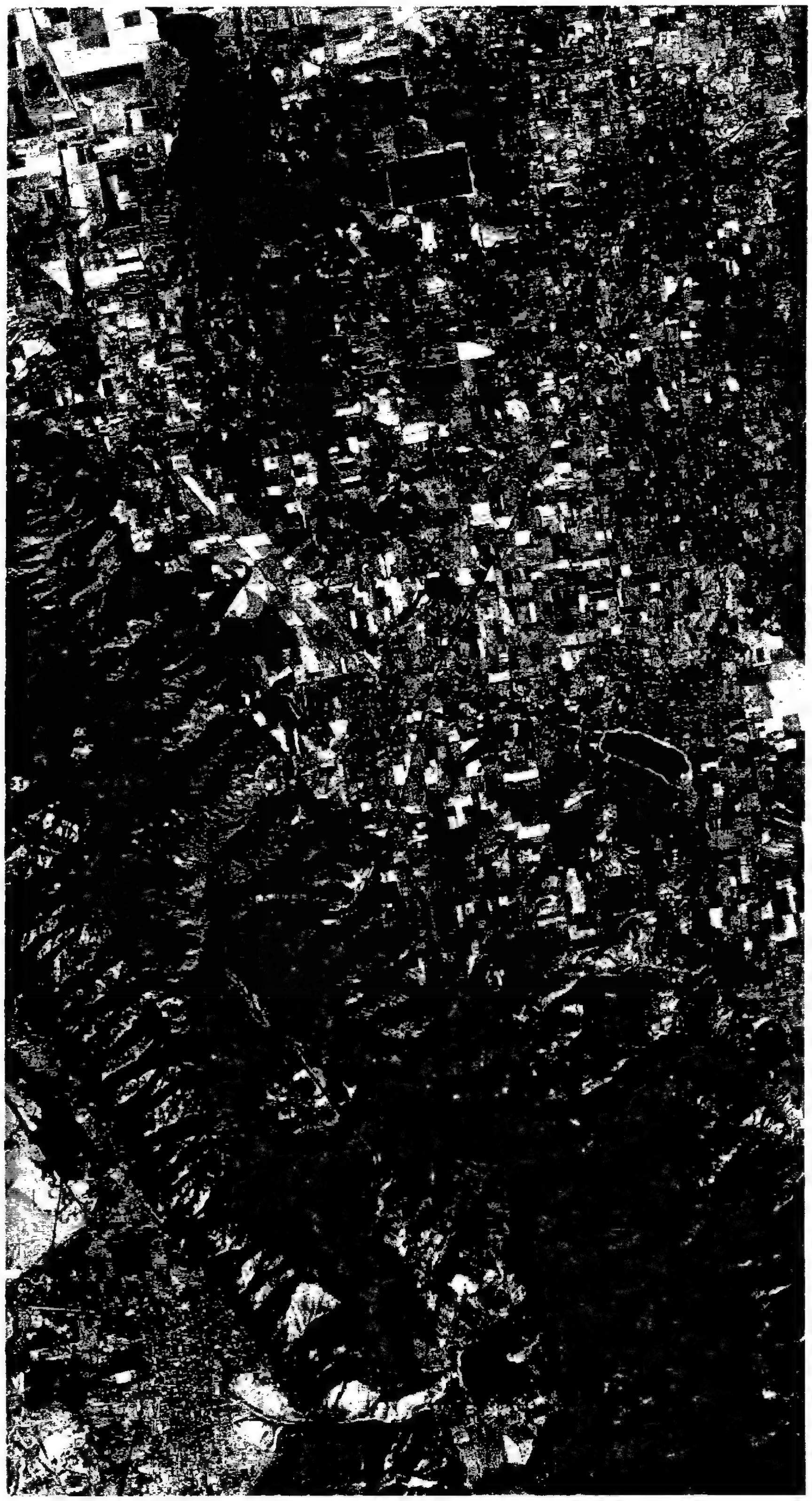


Fig. 2. Landsat bands 1, 4, 2.

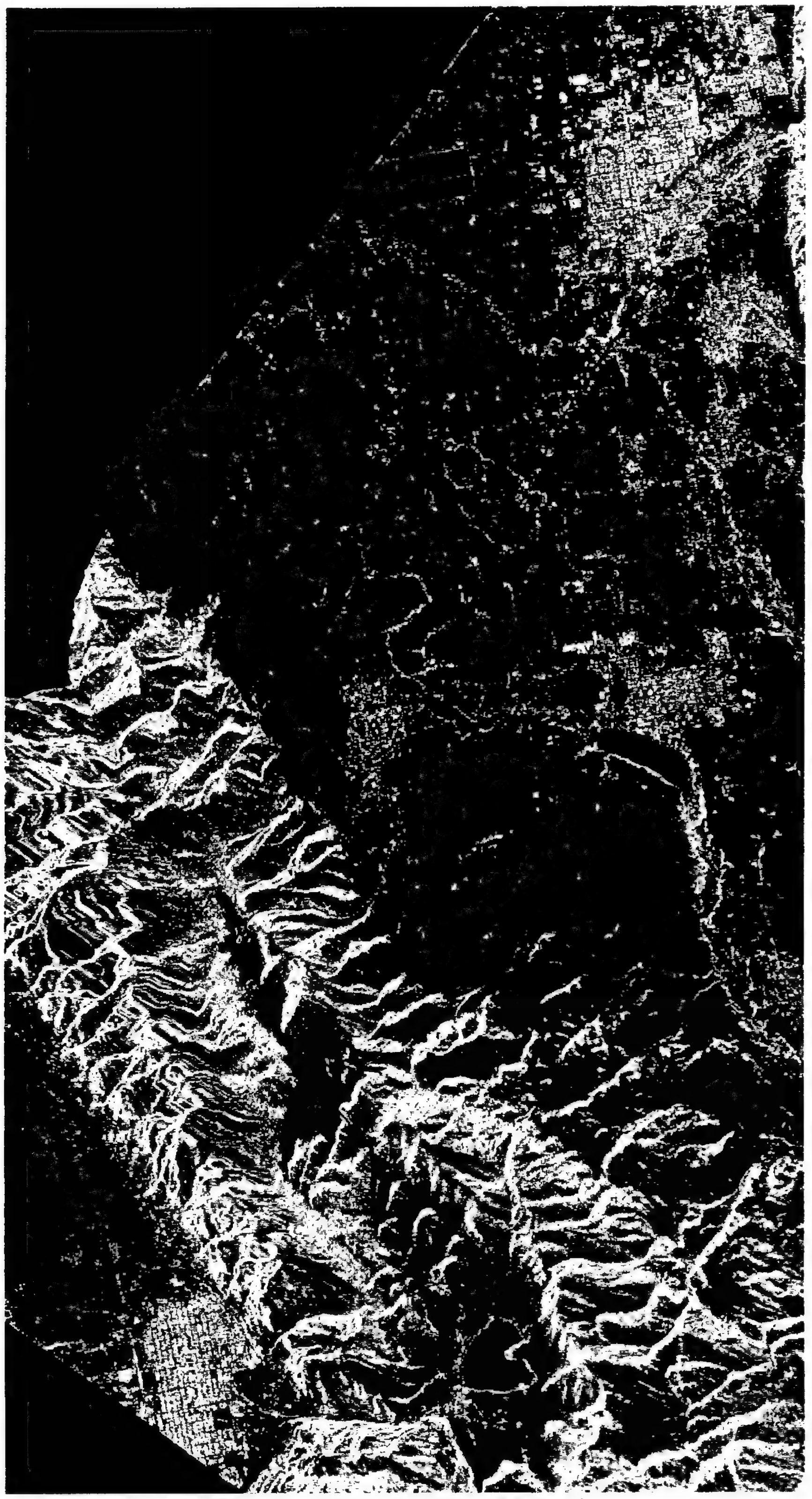


Fig. 3: SAR band C and band L

CHAPTER III

MULTIRESOLUTION PYRAMIDS

A. Image Structure and Multiresolution Pyramids

This study will use three techniques, described in this chapter, to represent an image over a range of scales. Other techniques are also presented here because they are necessary for understanding the three techniques used in the study. The techniques used to represent the images over a range of scales are called pyramids; the three of interest are RoLP, gradient, and morphological pyramids.

A complete image description can be obtained by studying an image structure over a range of scales. When we zoom in on an image, we clearly see the substructure; however, we lose the clarity of the outlines. On the other hand, when we zoom out to look at the entire picture, the scene loses detail. It logically follows that relevant details of an image can be observed only within a certain range of spatial resolution. If we focus on the small details, we lose focus of the big picture; on the other hand, if we zoom out to see the whole picture, it is difficult to discern the small details. Figure 4 illustrates this point. When zoomed out, the canopy and intake of the F-15 can be easily discerned; however, when zoomed in on the area outlined by the white box in the next image, the exact position of the canopy and intake edges are not as easily seen. For this reason, it is desirable to represent an image over a range of scales, depending upon the structural content.

A series of images progressively smaller in structural content can be created by repetitive application of a processing operator with a progressively increasing scale. This

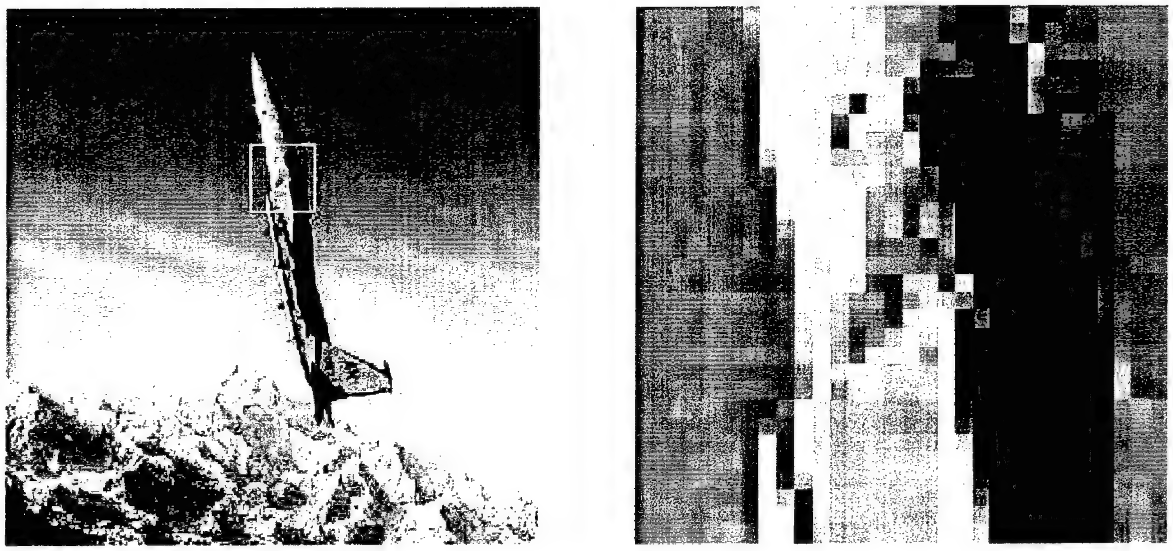


Fig. 4. Demonstration of necessity for multiple resolutions.

operator would eliminate details smaller than a certain size. This operator acts like a filter, just as sifting gravel through screens with different wire spacings filters gravel into different groups, dependent upon particle size. Repetitive application of this operator separates the image into scenes with different resolution of detail. By reducing the sample frequency and increasing the filter size, a hierarchical relation is generated. Reducing the sample frequency is the same as subsampling the image. A pyramid is a sequence of images in which each image is a filtered and subsampled copy of its predecessor [11]. The term "multiresolution pyramid" comes from the relationship where successive levels in a pyramid are reduced resolution copies of the input image [11-17].

The function that generates the next level of the pyramid could be called REDUCE since both the resolution and sample density are decreased. REDUCE would both filter and subsample the image. To create a pyramid starting with the source image as P_0,

$$\mathbf{P}_k = \text{REDUCE}(\mathbf{P}_{k-1}) \qquad \text{for } k = 1,2,\ldots n, \qquad (1)$$

where **n** is the number of levels in the pyramid.

Pyramid reconstruction to recover an image from its pyramid will need an EXPAND function because each level differs in sample density. EXPAND is defined as follows:

$$\mathbf{P}_{k-1} = \text{EXPAND}(\mathbf{P}_k) \qquad \text{for } k = n-1, n-2, \ldots 0, \qquad (2)$$

where n is the number of levels in the pyramid. Specific details of the EXPAND and REDUCE operators are dependent upon which types of pyramids are used. Techniques used to generate pyramids can be classified into two types: (1) linear and (2) morphological. Sections B, C, D, and E describe linear techniques, and Section F describes morphological techniques.

B. Gaussian and Laplacian Pyramids

As given in section A, each image in a pyramid is a low-pass filtered and subsampled copy of the previous image. This is beneficial for our purposes because it defines the image features in a set of images with different spatial resolutions by filtering. The most common linear low-pass filter used for pyramid generation is convolution with a Gaussian kernel. Pyramids formed using this technique are commonly referred to as Gaussian pyramids [11, 12, 14, 16, 17].

To create a Gaussian pyramid from an image **I**, assume $\mathbf{G}_k$ is the kth level of the pyramid. The bottom level of the pyramid, $\mathbf{G}_0$, equals **I**; and

$$\mathbf{G}_k = [\mathbf{w} * \mathbf{G}_{k-1}]\downarrow 2 \qquad \text{for } k > 0, \qquad (3)$$

where $[\ldots]_{\downarrow 2}$ denotes subsampling by two (The resultant image has .5 the x and y dimensions of the source image.), * is the convolution operator, and **w** is the Gaussian kernel. All of the elements of **w** must sum to equal 1; and, for simplicity, **w** is defined as separable, where $\mathbf{w} = \dot{\mathbf{w}} * \dot{\mathbf{w}}$. Convolution with the **w** matrix, as defined below, yields a Gaussian-similar curve; with **w** defined, $\dot{\mathbf{w}}$ can be derived. The **w** matrix defined in [12] is

$$\mathbf{w} = \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 25 & 16 & 4 \\ 6 & 25 & 36 & 25 & 6 \\ 4 & 16 & 25 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix} \left(\frac{1}{256} \right), \tag{4}$$

which yields a $\dot{\mathbf{w}}$ matrix defined as

$$\dot{\mathbf{w}} = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \left(\frac{1}{16} \right) \text{[12]}. \tag{5}$$

It should be noted that this value for **w** was used because of its Gaussian-like properties; however, **w** can be defined differently as long as it meets the criteria mentioned above.

Low-pass filtering and subsampling can be combined into a single function fitting the description of the REDUCE operator defined in the previous section. The REDUCE operator can be implemented as a function where each pixel in the new image is set to the weighted average of the corresponding area of pixels in the previous image. This process is much simpler than the previous process of convolution and subsampling (one-quarter the number of operations), yet it yields the same result. This is because the value returned by the weighted average is the same as the value returned by convolution.

From the weighted average method, the kth level of the pyramid is defined on a pixel by pixel basis as

$$\mathbf{P}_k(i,j) = \sum_{m,n=-2}^{2} w(m,n)\mathbf{P}_{k-1}(2i+m, 2j+n) \qquad \text{for } \mathbf{k} > 0. \tag{6}$$

The weighting function *w(m,n)* is separable,

$$w(m,n) = w'(m)w'(n).$$

The function w' is normalized,

$$\sum_{m=-2}^{2} w'(m) = 1, \tag{7}$$

and is symmetric,

$$w'(i) = w'(-i) \qquad \text{for i = 0, 1, 2.} \tag{8}$$

All of these constraints are satisfied when *w'* is defined as follows [11]:

$$w'(0) = a; \tag{9}$$
$$w'(1) = w'(-1) = 1/4; \tag{10}$$
$$w'(2) = w'(-2) = 1/4 - a/2. \tag{11}$$

The value of a used in this study was 0.4 because of its Gaussian-like properties [11, 12, 14]. The reader should note that $\mathbf{w}$ and $\dot{\mathbf{w}}$ are matrices used in convolution, whereas *w* and *w'* are functions used in the weighted average.

Taking the difference between two levels in a Gaussian pyramid results in a band-pass filtered image. In order to take the difference between the two levels, however, the level of lower resolution must be expanded to the same size as the image at the higher resolution. If this is done for every level of a Gaussian pyramid, a Laplacian pyramid is created. Let $\tilde{\mathbf{L}}_k$ be the kth level of the Laplacian pyramid, defined as

$$\tilde{\mathbf{L}}_k = \mathbf{G}_k - 4\mathbf{w} * [\mathbf{G}_{k+1}]\uparrow_2, \tag{12}$$

using the convolution notation, where $[\ldots]\uparrow_2$ indicates upsampling by two [12]. When upsampling, rows and columns of zeros are added between the existing rows and columns. Convolution with **w** interpolates values for the columns and rows of zeros.

The Laplacian pyramid can also be defined using the weighted average method as

$$\tilde{\mathbf{L}}_k = \mathbf{G}_k - \text{EXPAND}(\mathbf{G}_{k+1}) \text{ [11, 14].} \tag{13}$$

For linearly filtered pyramids, EXPAND can be defined as

$$\mathbf{P}_k(i, j) = 4 \cdot \sum_{m,n=-2}^{2} w(m,n)\mathbf{P}_{k-1}\left(\frac{i+m}{2}, \frac{j+n}{2}\right), \tag{14}$$

where only integer coordinates contribute to the sum. It is noted that in the Laplacian pyramid $\tilde{\mathbf{L}}_n = \mathbf{G}_n$, namely, the top level of the Laplacian pyramid, is the same image as the top level of the Gaussian pyramid.

The pyramid algorithm reduces the filter band limit by an octave from level to level. The Laplacian pyramids are the equivalent of a band-pass filter with a bandwidth equal to the span between octaves. Filtering an image has the effect of blurring. This is because sharp edges are composed of high-frequency components, and the low-pass filter attenuates the high spatial frequencies in the image. Images higher in the pyramid structure have fewer high-frequency components than images in the lower part of the pyramid. Therefore, when expanded to the size of an image lower in the pyramid, the image higher in the pyramid appears blurry.

Reconstructing an image from its Laplacian pyramid is straightforward. The original image is recovered by expanding the highest level of the pyramid and adding that to the next highest level repeatedly, until no levels are left. This is mathematically expressed as

$$\hat{\mathbf{G}}_n = \tilde{\mathbf{L}}_n \tag{15}$$

and $$\hat{\mathbf{G}}_k = \tilde{\mathbf{L}}_k + \text{EXPAND}(\hat{\mathbf{G}}_{k+1}) \qquad \text{for } k = n-1, n-2, \ldots 0. \tag{16}$$

This yields an exact reconstruction of the source image, $\hat{\mathbf{G}}_0 = \mathbf{G}_0$.

C. *Filter-Subtract-Decimate Laplacian Pyramids*

A level of the filter-subtract-decimate (FSD) Laplacian pyramid is defined as the difference between the Gaussian level and the filtered copy of the Gaussian level prior to subsampling for the next Gaussian level. For this reason, the REDUCE function as defined will not work with FSD Laplacian pyramids because REDUCE filters and subsamples at the same time. Let $\mathbf{L}_k$ be the kth level of the FSD Laplacian pyramid, which is mathematically defined as

$$\mathbf{L}_k = \mathbf{G}_k - \mathbf{w} * \mathbf{G}_k = [\mathbf{1} - \mathbf{w}] * \mathbf{G}_k, \tag{17}$$

where **1** is a matrix of the same dimensions of **w** where all values are zero except for the center value, which is one. In the case of a 3-by-3 matrix,

$$\mathbf{1} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}. \tag{18}$$

Inspection of the process of forming the image in the Laplacian pyramid, $\tilde{\mathbf{L}}_k$, reveals that the Gaussian image, $\mathbf{G}_k$, is sequentially convolved with **w**, subsampled, upsampled, convolved with **w** again, and then subtracted from itself. Refer to equations 3 and 12. However, if the resampling steps are skipped, the results are only slightly different. We can, therefore, say that

$$\begin{aligned}\tilde{\mathbf{L}}_k &= \mathbf{G}_k - 4\mathbf{w} * [[\mathbf{w} * \mathbf{G}_k]\downarrow_2]\uparrow_2 \approx \mathbf{G}_k - \mathbf{w} * \mathbf{w} * \mathbf{G}_k \\ &= [1 - \mathbf{w} * \mathbf{w}] * \mathbf{G}_k \\ &= [1 + \mathbf{w}] * [1 - \mathbf{w}] * \mathbf{G}_k.\end{aligned} \tag{19}$$

Hence, the FSD Laplacian pyramid can be converted to a Laplacian pyramid through the following conversion:

$$\tilde{\mathbf{L}}_k \approx [1 + \mathbf{w}] * \mathbf{L}_k. \tag{20}$$

Pyramid reconstruction to obtain the original image follows by converting the FSD Laplacian pyramid into a Laplacian pyramid and then reconstructing the Laplacian pyramid [12]. Note that this reconstruction is not an exact replica of the original.

D. Ratio of Low-Pass Pyramids

The ratio of low-pass (RoLP) pyramid gets its name from the relationship that exists between successive levels of the pyramid. RoLP pyramids are very similar to Laplacian pyramids; instead of taking the difference between levels of a Gaussian pyramid, the RoLP pyramid takes the ratio between levels of a Gaussian pyramid. The RoLP pyramid, $\mathbf{R}_k$, is mathematically defined as

$$\mathbf{R}_k = \frac{\mathbf{G}_k}{\text{EXPAND}(\mathbf{G}_{k+1})} \qquad \text{for } k = n-1, n-2, \ldots 0 \tag{21}$$

and
$$\mathbf{R}_n = \mathbf{G}_n . \tag{22}$$

Every level in the RoLP pyramid is the ratio of two successive levels in the Gaussian pyramid.

Let $\hat{\mathbf{G}}_0$ be the image reconstructed from the RoLP pyramid. The reconstruction process is the inverse of the construction process.

$$\hat{\mathbf{G}}_n = \mathbf{R}_n , \tag{23}$$

and $$\hat{\mathbf{G}}_k = \mathbf{R}_k \cdot \text{EXPAND}(\mathbf{G}_{k+1}) \quad \text{for } k = n-1, n-2, \ldots 0. \tag{24}$$

The reconstruction process for RoLP pyramids is exact, or in other words, $\hat{\mathbf{G}}_0 = \mathbf{G}_0$. There is also a contrast-enhanced RoLP pyramid described in [17] that was not used in this study.

E. Gradient Pyramids

The term "gradient pyramid" is a misnomer, because a gradient pyramid is actually a collection of four pyramids. Let $\mathbf{D}_{km}$ represent the kth level and mth orientation gradient pyramid image for an image **I**. $\mathbf{D}_{km}$ is obtained from convolving $\mathbf{G}_k$ with a gradient filter $\mathbf{d}_m$, also called an "oriented second derivative filter."

$$\mathbf{D}_{km} = \mathbf{d}_m * [\mathbf{G}_k + \dot{\mathbf{w}} * \mathbf{G}_k], \tag{25}$$

where

$$\mathbf{d}_1 = \begin{bmatrix} 1 & -1 \end{bmatrix}; \tag{26}$$

$$\mathbf{d}_2 = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \frac{1}{\sqrt{2}}; \tag{27}$$

$$\mathbf{d}_3 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}, \tag{28}$$

and $$\mathbf{d}_4 = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \frac{1}{\sqrt{2}}. \tag{29}$$

Reconstruction of an image from a gradient pyramid requires Laplacian and FSD Laplacian pyramids as intermediate steps. Each gradient pyramid level $\mathbf{D}_{km}$ is converted to a corresponding second derivative pyramid or oriented Laplacian level $\vec{\mathbf{L}}_{km}$.

$$\vec{\mathbf{L}}_{km} = -\frac{1}{8} \mathbf{d}_m * \mathbf{D}_{km}. \tag{30}$$

An FSD Laplacian pyramid, $\mathbf{L}_k$, is then formed by summing the oriented Laplacian pyramids.

$$\mathbf{L}_k = \sum_{m=1}^{4} \vec{\mathbf{L}}_{km}. \tag{31}$$

Reconstruction is then completed by following the same steps required to reconstruct an FSD Laplacian pyramid [12] as described in Section C of this chapter. Since the FSD Laplacian pyramid is used, composite image reconstruction is approximate, or $\hat{\mathbf{G}}_0 \approx \mathbf{G}_0$.

F. Mathematical Morphology Pyramids

Mathematical morphology is the examination of the geometric structure of an image by probing its microstructure with certain elementary forms called structuring elements [15]. Linear filters alter the object intensities and, therefore, estimate the location of contours. Morphological filters, on the other hand, remove details without adding grayscale bias and, therefore, are nicely suited to shape extraction. This section contains a brief summary of mathematical morphology and its operations. For more information on any of the topics in this section, the reader is referred to [11, 15, 22, 23].

Mathematical morphology is based upon set operations. When used with images, the first set is considered to be the image; the second set is the structuring element. The sets are representative of shapes that are manifest in the image. Selection of the structuring element is an important part of morphological operations. The smaller the structuring element, the smaller the details that are filtered or modified. When generating a pyramid by successively increasing the size of the structuring element or by reducing the image size, successively larger image details are filtered out, once again leading to the multiple resolutions. Also, when reconstructing the image, the positional accuracy cannot be better than the radius of the structuring element.

The building blocks of morphology are the dilation and erosion operators. Dilation is a transform that combines two sets using vector addition of set elements. If **A** and **B** are sets, the dilation of **A** by **B** is the set of all possible vector sums of pairs of elements, one from **A** and one from **B**. Dilation is denoted by the $\oplus$ symbol and is mathematically defined as

$$\mathbf{A} \oplus \mathbf{B} = \{\mathbf{c} \in \mathbf{E}^N | \mathbf{c} = \mathbf{a} + \mathbf{b} \text{ for some } \mathbf{a} \in \mathbf{A} \text{ and } \mathbf{b} \in \mathbf{B}\}, \tag{32}$$

where **A** and **B** are the sets in Euclidean N-space ($\mathbf{E}^N$) with elements **a** and **b,** respectively. The complexity of image dilation and erosion is similar to that of convolution. To easily conceptualize dilation, picture a straight line; after dilation, this line becomes wider.

Erosion is the morphological dual of dilation. The erosion transform combines two sets using the vector subtraction of set elements. If **A** and **B** are sets in Euclidean N-space, erosion of **A** by **B** is the set of all elements **x** for which $\mathbf{x} + \mathbf{b} \in \mathbf{A}$ for every $\mathbf{b} \in \mathbf{B}$. Erosion is denoted by the Θ symbol and is mathematically defined as

$$\mathbf{A} \Theta \mathbf{B} = \{\mathbf{x} \in \mathbf{E}^N | \mathbf{x} + \mathbf{b} \in \mathbf{A} \text{ for every } \mathbf{b} \in \mathbf{B}\}, \tag{33}$$

where **A** and **B** are the sets in Euclidean N-space with elements **a** and **b,** respectively. Once again, for easy conceptualization, picture a thick line; everywhere the line is thick enough to contain the structuring element, it becomes thinner. If this line is dilated and then eroded, it would appear unchanged because everywhere dilation made it wider, erosion made it thinner.

Conceptually, an erosion shrinks image features, and a dilation expands image features. Figure 5 shows the erosion and dilation of a box. Note how the box walls look thinner after dilation and thicker after erosion; if this seems the opposite of the expected result, reconsider the expected result focusing on white areas as the boxes. Knowing that morphology is based on set theory, consider the color black the empty set. The black walls are areas where no information exists. (Pixel intensity = 0 for red, green, and blue pixels.) Now consider that the color white is the full set; this would be areas containing all three colors: red, green, and blue. Where blue is seen, only blue is present; where red is seen, only red is present; and in areas where a pinkish-purple color is seen (only in Figure 5 (b)), both red and blue are present.

The source image has four areas of concern: the black box around the outside where no pixels have values greater than zero; the white area between the black and blue boxes where red, green, and blue all have pixel values greater than zero; the blue area

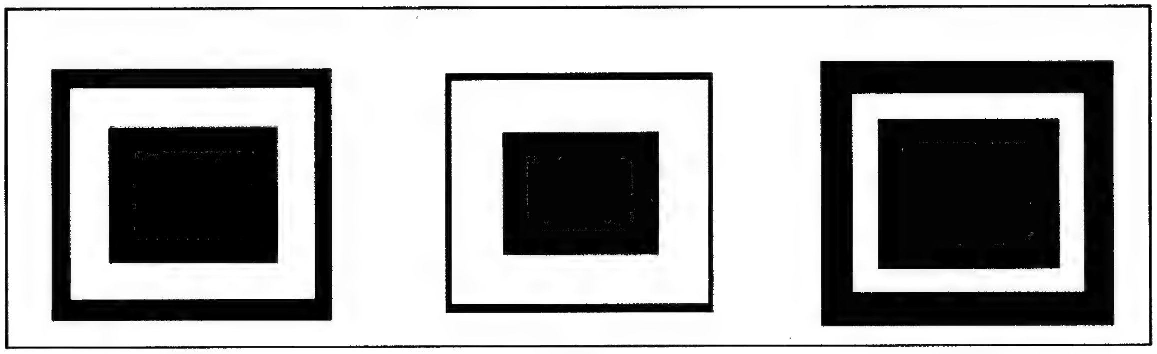

(a) (b) (c)

Fig. 5. Example of erosion and dilation operators using a 10x10 brick structuring element. (a) Source image. (b) Source image after dilation. (c) Source image after erosion.

where only blue pixels have values greater than zero; and the red area where only the red pixels have values greater than zero.

After dilation, the white walls become thicker, making the black walls thinner. Likewise, both the red walls and the blue walls expand, causing a new area where both red and blue values exist simultaneously without green. Although the blue and red box appear smaller in size, they have actually grown; we do not see their growth because, in the areas where they have grown, other pixel values exist so we either see white or else the combination of blue and red, which is the pinkish-purple color in Figure 5 (b).

When erosion occurs, the white walls become thinner, making the black walls bigger. When the color boxes are viewed, it is easy to see, by looking at the size of the red box, that erosion makes the boxes smaller. The black box around the red box in Figure 5 (c) is an area where both the red box and the blue box have been eroded and no image data are left, hence the empty set, or black. For more information about the erosion and dilation of a set or the properties associated with erosion and dilation, refer to [22, 23].

The steps for creating a morphological pyramid are the same as creating a Gaussian pyramid. To generate the next pyramid level, the current image is filtered and subsampled.

Morphological filters are sequences of morphological operations that have special properties with respect to the shapes in the image. Morphological filters are idempotent and increasing. Idempotent means that successive applications of the filter leave the result unchanged after the first time the filter has been applied, similar to the application of a linear band-pass filter to a signal. Increasing means that the operations maintain inclusion relationships on the images they transform [15]; or if set **A** is a subset of set **B,** then **A** filtered by **F** is a subset of **B** filtered by **F**.

The simplest morphological filters are the opening and closing transformations. An opening is an erosion followed by a dilation and is denoted by the $\circ$ symbol. A closing is a dilation followed by an erosion and is denoted by the $\bullet$ symbol. Mathematically, opening and closing are defined as follows:

$$\mathbf{A} \circ \mathbf{f} = (\mathbf{f} \Theta \mathbf{A}) \oplus \mathbf{A}\,; \tag{34}$$
$$\mathbf{A} \bullet \mathbf{f} = (\mathbf{f} \oplus \mathbf{A}) \Theta \mathbf{A}\,, \tag{35}$$

where **A** is the image and **f** is the structuring element. These filters are also low-pass because they attenuate high-frequency fluctuations between the set and its complement. Openings and closings are considered dual operators because what one does to the foreground, the other does to the background. To ensure that the foreground and background of an image are treated the same, openings and closings usually follow one another. Morphological filters are most often a combination of openings and closings or closings and openings. For this study, we will use F to represent a closing-opening filter:

$$\mathrm{F} = (\mathbf{A} \bullet \mathbf{f}) \circ \mathbf{f}\,. \tag{36}$$

With morphological filters defined, we can describe the generation of morphological pyramids in a manner similar to the generation of Gaussian pyramids. Let **I** represent the original image. The base of the pyramid, $\mathbf{M}_0$, needs to be a morphologically filtered copy of **I;** the filter used to generate $\mathbf{M}_0$ will determine what filters are to be used for pyramid reconstruction. We will again define the REDUCE function, which will, in this case, morphologically filter the image and then subsample.

To generate the next level in the pyramid, REDUCE will be applied the current level. Let $\mathbf{M}_0$ be the base of the pyramid, and

$$\mathbf{M}_k = \text{REDUCE}(\mathbf{M}_{k-1}) \qquad \text{for } 1 \leq k \leq n, \tag{37}$$

where n is the depth of the pyramid and REDUCE is $F(\mathbf{M}_{k-1}) \downarrow_2$. From here, a difference pyramid similar to the linear Laplacian pyramid can be constructed where

$$\mathbf{D}_n = \mathbf{M}_n \tag{38}$$

and

$$\mathbf{D}_k = \mathbf{M}_k - \text{EXPAND}(\mathbf{M}_{k+1}) \qquad \text{for } k = n-1, n-2, \ldots 0. \tag{39}$$

Here, EXPAND is defined as upsampling followed by a closing. REDUCE and EXPAND can use any filter; however, the user must ensure that the filters used in the EXPAND function complement the filters used in the REDUCE function [22, 23]. If the initial filtering of the source image **I** is a closing, then a minimal reconstruction should be performed; and the EXPAND function should use the closing filter. On the other hand, if the initial filtering of the source image **I** is an opening, then a maximal reconstruction should be performed; and the EXPAND function should use the dilation filter.

Pyramid reconstruction is once again straightforward. The reconstructed image, $\hat{\mathbf{M}}_0$, can be obtained by setting

$$\hat{\mathbf{M}}_n = \mathbf{D}_n \tag{40}$$

and

$$\hat{\mathbf{M}}_k = \mathbf{D}_k + \text{EXPAND}(\hat{\mathbf{M}}_{k+1}) \qquad \text{for } k = n-1, n-2, \ldots 0. \tag{41}$$

$\hat{\mathbf{M}}_n$ is an exact reconstruction of the pyramid if no other pyramid processing has occurred to either the morphological pyramid or the difference pyramid.

CHAPTER IV

PYRAMID FUSION TECHNIQUES

A. *Pyramid Fusion Overview*

This study will use three of the pyramid techniques described in the previous chapter to combine or fuse two source images into a single composite image. In order to do this, we must define a way to fuse two pyramids into a single pyramid. This chapter will introduce the fusion techniques described in current literature that were used in this study.

Pyramids are simply a convenient way to represent an image over a range of spatial resolutions. By combining the images at each level of the pyramid, the composite image, formed by pyramid reconstruction, will have consistency over all resolutions.

When fusing two pyramids, each of the levels of the pyramids is fused into a composite level, resulting in a composite pyramid. Refer to Figure 1. Once the composite pyramid is formed, the fused image of the source images is generated, employing the pyramid reconstruction techniques associated with the technique used to generate the source pyramids. For example, if Laplacian pyramids **A** and **B** were generated from two source images, the composite image resulting from the fusion of pyramids **A** and **B** would be reconstructed from the composite pyramid **C,** using the techniques described in Chapter III to reconstruct an image from its Laplacian pyramid. Each level of the composite pyramid is defined as

$$\mathbf{C}_k = \mathrm{FUSE}(\mathbf{A}_k, \mathbf{B}_k) \qquad \text{for } k = n, n-1, n-2, \ldots 0, \qquad (42)$$

and n is the number of levels in the pyramid. FUSE is a function that converts the two images into the composite, using a fusion algorithm.

B. Selection and Averaging Fusion Techniques

When combining two images, the untrained, intuitive approach might be to average the pixels from the source images to obtain the value for the corresponding pixel in the composite image. This approach is undesirable because features that appear in one image and not the other will show up in the composite at a reduced contrast or will appear superimposed on features from the other image, much like when camera film is double exposed. To avoid the fusion problems by averaging, the composite image can be obtained by selecting pixels from either of the source images.

The simplest pixel selection technique for RoLP pyramids is to use the local area contrast to determine which pixel to select. The contrast of a given pixel is defined as the ratio of the difference in pixel intensity to area intensity to the area intensity, or as follows:

$$Contrast(i,j) = \frac{\mathbf{L}(i,j) - \mathbf{L}_b(i,j)}{\mathbf{L}_b(i,J)} = \frac{\mathbf{L}(i,j)}{\mathbf{L}_b(i,j)} - 1, \tag{43}$$

where $\mathbf{L}$ is the luminance at (i, j), or simply the pixel intensity, and $\mathbf{L}_b$ is the background luminance for that area. Note that the ratio of $\mathbf{L}$ to $\mathbf{L}_b$ is the value in the RoLP pyramid; hence, to get the contrast for a certain pixel value, simply subtract one from the RoLP value.

The human eye detects contrast very well. When taking advantage of this fact, pixels of maximum contrast are selected from each source image to form the composite. The logic behind this method is that by selecting details of maximum contrast, the fused image will provide better details for human analysis. The contrast version of the FUSE function is implemented on a pixel-by-pixel basis, as follows:

$$\mathbf{C}(i,j) = \begin{cases} \mathbf{A}(i,j), & \text{when } \left(\left|\mathbf{A}(i,j)-1\right| \geq \left|\mathbf{B}(i,j)-1\right|\right) \\ \mathbf{B}(i,j), & \text{otherwise} \end{cases} \quad [11, 16, 17]. \quad (44)$$

However, another reference implemented the contrast fusion in a slightly different way, as follows:

$$\mathbf{C}(i,j) = \begin{cases} \mathbf{A}(i,j), & \text{when } \left(\left|\mathbf{A}(i,j)\right| \geq \left|\mathbf{B}(i,j)\right|\right) \\ \mathbf{B}(i,j), & \text{otherwise} \end{cases} \quad [13]. \qquad (45)$$

It is interesting that it was implemented this way because, without subtracting one from the RoLP pyramid value, it is not based on the area contrast anymore.

C. *Hybrid Averaging and Selection Fusion*

The problem with contrast fusion is that it is susceptible to noise. Noisy images are typically of higher contrast. Using a contrast image fusion technique would, therefore, result in a composite image with more noise. The combination of both selection and averaging to generate a composite image was proposed by [12]. For this method to work, a metric is necessary to indicate when to use selection and when to use averaging; this metric is called "match." When the match measure is a value that corresponds to using selection, it is necessary to know from which image to select. This requires a saliency measure. When the two images are distinctly different, the composite image should select the most salient component. However, when the two images are similar, the composite image should contain the mean of the two source images. This technique makes it possible to reduce noise without double exposure artifacts [12].

A good measure of pattern salience is pixel intensity. To obtain the salience for a given sample, the individual pixel intensity can be used. Alternatively, the average pixel intensity of a small area near the sample can be used. The mean pixel intensity can be mathematically defined as a weighted average over an area, **p,** using $\mathbf{S}_k(i,j)$ as the saliency measure of the pixel at *(i,j)* for the image *k***,** as follows:

$$\mathbf{S}_k(i,j) = \sum_{i',j'=-m}^{m} \mathbf{p}(i',j')\mathbf{P}_k(i+i',j+j'), \qquad (46)$$

where *m* can range from 0, which would be the individual sample, to 2, which would include the 5x5 area surrounding the sample. The value of $\mathbf{p}(i',j')$ is weight of the sample within the area or neighborhood **p**; samples closer to the *(i, j)* position have higher weight values. The function **p** serves the same purpose here as does the function *w'* in Chapter III, Section B. The value returned by $\mathbf{P}_k(i+i',j+j')$ returns the pixel value inside the image at location *(i, j)*, offset by *i'* and *j'*; it can simply thought of as $\mathbf{P}_k$ being the matrix indexed by *(i+i', j+j')*.

Relative pixel intensity between the two images can be used as a measure of match. Alternatively, correlation is well-suited as a measure of match. We can mathematically define the match of image **A** to **B,** $\mathbf{M}_{AB}$, within the area **p,** as the normalized correlation between image **A** and **B**, or

$$\mathbf{M}_{AB}(i,j) = \frac{2\sum_{i'j'=-m}^{m} \mathbf{p}(i'j')\mathbf{A}(i+i',j+j')\mathbf{B}(i+i',j+j')}{\mathbf{S}_A(i,j)+\mathbf{S}_B(i,j)}, \text{ [12]} \qquad (47)$$

where *m* can range from 0 to 2, depending upon the desired area of **p** (0 for an individual point, 2 for a 5x5 matrix). The values returned for match are between –1 and 1. Values close to zero indicate low correlation, and values close to –1 or 1 indicate high correlation.

Each level of the pyramid can now be fused by testing the match metric between the two images at the given level. If the match metric is low at a given position, then the sample from the source image with the highest salience is copied to the composite image. If the match metric is high, the pixels from the source images are averaged and copied to the composite image. This combination technique can be implemented as a weighted mean in which the weights depend upon the match and saliency measures. The function FUSE, as described above, can then be described as follows:

$$\mathbf{C}(i,j) = w_A(i,j)\mathbf{A}(i,j) + w_B(i,j)\mathbf{B}(i,j)\,, \tag{48}$$

where w_A and w_B are the assigned weights to the source images **A** and **B**, respectively; and $\mathbf{w}_A + \mathbf{w}_B = 1$.

There are two common ways to implement the weighting functions w_A and w_B. One way is to select some threshold, α, for the match metric. We can set w_A and w_B to 1 and 0 or 0 and 1, respectively, when the match is below α, or we can set w_A and w_B to .5 and .5 when the match is above α. This would be mathematically written as

$$\begin{aligned}
&\text{if } \alpha < \text{threshold}\\
&\quad w_A = .5 \text{ and } w_B = .5,\\
&\text{else if } \mathbf{S}_A \geq \mathbf{S}_B\\
&\quad w_A = 1 \text{ and } w_B = 0, \text{ else } w_A = 0 \text{ and } w_B = 1.
\end{aligned} \tag{49}$$

This technique would require extensive testing to ascertain an appropriate α for each type of image used and does not allow for a gradual change from selection to averaging.

Another common way to implement the weight functions is to assign the weight value based on a linear transition between 1 and 0 as follows: If $\mathbf{M}_{AB} \le \alpha$, then $w_{min} = 0$ and $w_{max} = 1$; otherwise,

$$w_{min} = \frac{1}{2} - \frac{1}{2}\left(\frac{1-\mathbf{M}_{AB}}{1-\alpha}\right), \tag{50}$$

and

$$w_{max} = 1 - w_{min}. \tag{51}$$

$$w_A = w_{max} \text{ and } w_B = w_{min} \text{ if } \mathbf{S}_A \ge \mathbf{S}_B, \text{ else } w_A = w_{min} \text{ and } w_B = w_{max}. \tag{52}$$

Here, the larger weight is assigned to the source image with the higher saliency value. This latter approach allows a gradual change from selection to averaging and was the method used in this study because it offers the benefits of both selection and averaging over a range of match values rather than simply selecting one method or the other. This is the same method used in [12].

CHAPTER V

FUSED IMAGERY AND RESULTS

A. *Fusion Approach*

In this study, six multiresolution image fusion functions are applied using the pyramid and fusion techniques described in Chapters III and IV to demonstrate the feasibility of image fusion in remote sensing imagery. The first three functions have been described in the current literature. The other three functions are combinations of pyramids and fusion techniques that have not been described in current literature. Appendix C contains computer programs used to implement the six combinations for multiresolution image fusion. For simplicity, the images presented in this chapter are only selected subimages of the images presented in Chapter II.

Three of the basic pyramid types described in Chapter III for image decomposition are employed in this study. The pyramids used in the fusion functions are the RoLP, gradient, and morphological pyramids. (The other pyramids in Chapter III were discussed because understanding them is necessary in order to understand the three pyramids that were used.) These three pyramid techniques can be combined with the two fusion techniques in six possible ways. The three combinations described in the current literature are: (1) using a RoLP pyramid for image decomposition with contrast fusion for image merging [11, 16]; (2) using a gradient pyramid for image decomposition with hybrid averaging and selection fusion for image merging [12]; and (3) using a morphological pyramid for image decomposition with contrast fusion for image merging [13]. The three not discussed in current literature are: (1) using a RoLP pyramid for image decomposition with hybrid averaging and selection fusion for image merging; (2) using a gradient pyramid for image decomposition with contrast fusion for image merging; and (3) using a morphological pyramid for image decomposition with hybrid averaging and selection fusion for image merging.

In previous studies, composite Landsat and SAR images have been created by assigning specific bands, or ratios of bands, to specific pixel colors. Images that have pixels representing both SAR and Landsat data will be called "hybrid SAR and Landsat images" or simply "hybrid images." Hybrid images can be very useful, as presented in [5]; however, they limit the number of bands that can be viewed to three. By applying multiresolution fusion functions to source images, it is the purpose of this study to make it possible to effectively view images that contain information from more than three spectral bands.

The remainder of this thesis presents the results and conclusions of applying the six fusion functions to test images. Fusion of remotely sensed data can take on several forms. In the following, Section B presents the results of fusing SAR images with Landsat images to obtain information about the fusion functions. Section C discusses fusion of hybrid images to obtain useful information as well as hybrid image fusion applications. Finally, in Section D, the factors relevant to composite image quality are considered.

B. Fusing SAR and Landsat Images to Evaluate Fusion Functions

In this section, fusing imagery of the Mantua, Utah, area is reported. This set of imagery is used because it contains mountainous and agricultural areas as well as urban areas. A list of the figures and parameters used in this section is given in Table III. The first source image, Figure 6 (a), was obtained from the Landsat-4 satellite on 25 September 1994; the RGB pixel values are from bands 4, 2, and 1, respectively. The second source image, Figure 6 (b), was obtained from the SIR-C SAR instrument flown on STS-68 on 2 October 1994; the RGB pixel values are from bands L, C, and C, respectively.

The results presented in Figures 7, 8, and 9 demonstrate that the gradient and RoLP pyramids with the hybrid fusion technique, Figures 7 (b) and 8 (b), provide the composite images with the most well integrated fusion of source image features. The

TABLE III
LIST OF FIGURES IN CHAPTER V, SECTION B

Figure	Area	Description
6 (a)	Mantua	Landsat imagery of Mantua, bands = (4, 2, 1).
6 (b)	Mantua	SAR imagery of Mantua, bands = (L, C, C).
7 (a)	Mantua	Selection fusion of 6 (a) and 6 (b) using a four level Gradient pyramid with a Gaussian kernel of a = 0.4.
7 (b)	Mantua	Hybrid fusion of 6 (a) and 6 (b) using a four level Gradient pyramid with a Gaussian kernel of a = 0.4 and a match threshold of $\alpha = 0.1$.
8 (a)	Mantua	Selection fusion of 6 (a) and 6 (b) using a four level RoLP pyramid with a Gaussian kernel of a = 0.4.
8 (b)	Mantua	Hybrid fusion of 6 (a) and 6 (b) using a four level RoLP pyramid with a Gaussian kernel of a = 0.4 and a match threshold of $\alpha = 0.1$.
9 (a)	Mantua	Selection fusion of 6 (a) and 6 (b) using a four level Morphological pyramid with a 2x2 brick as a structuring element.
9 (b)	Mantua	Hybrid fusion of 6 (a) and 6 (b) using a four level Morphological pyramid with a match threshold of $\alpha = 0.1$ and a 2x2 brick as a structuring element .
10 (a)	Mantua	Subsection of Figure 7 (b) enlarged for detail.
10 (b)	Mantua	Subsection of Figure 8 (b) enlarged for detail.

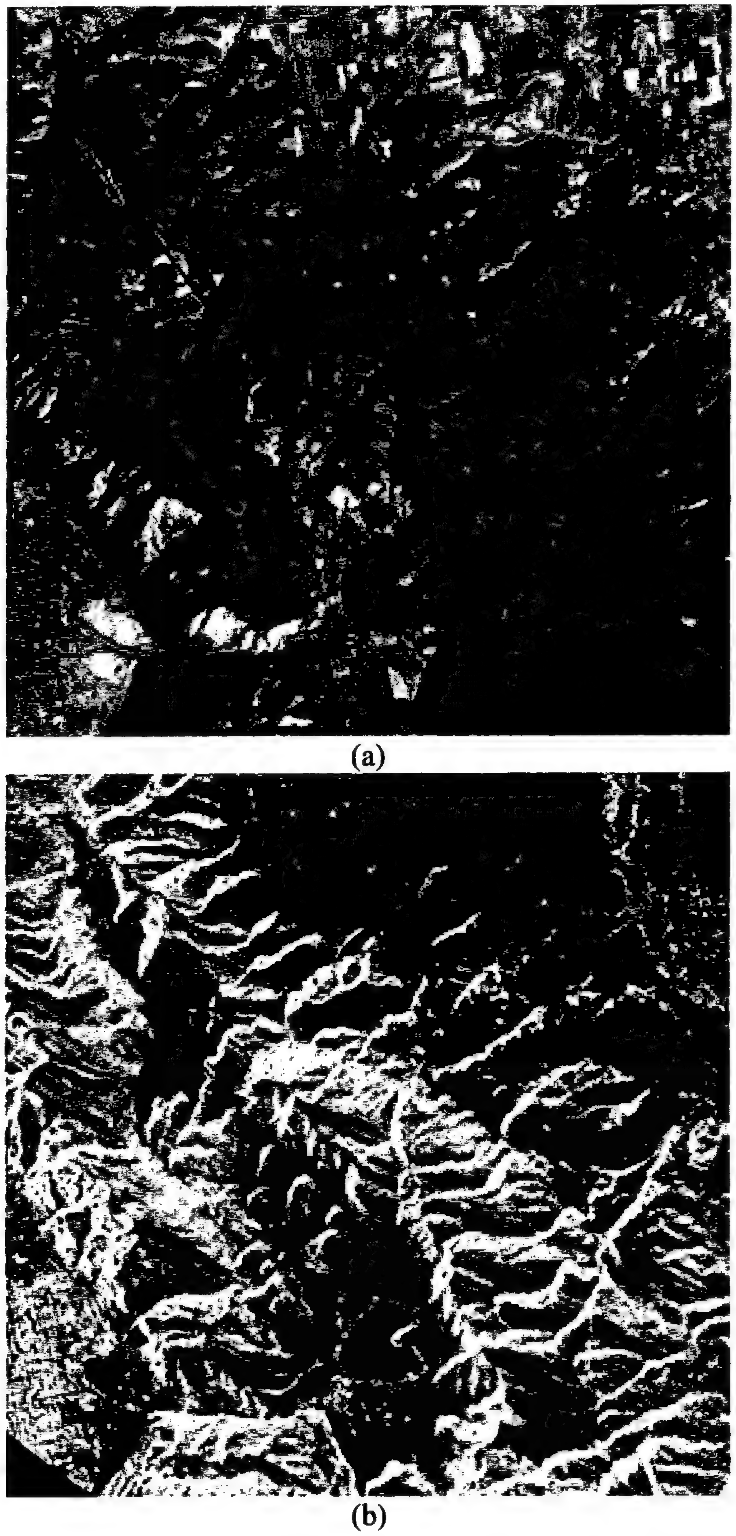
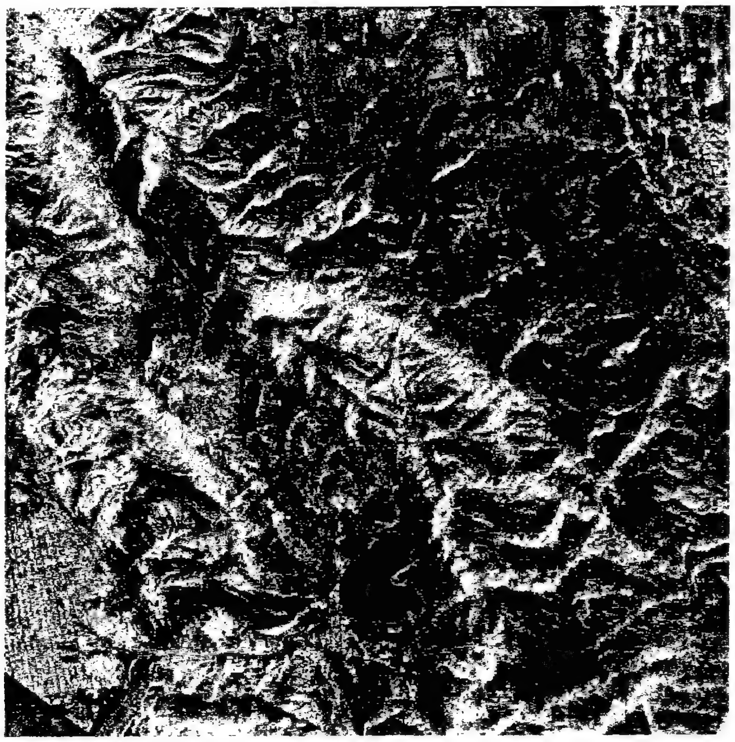
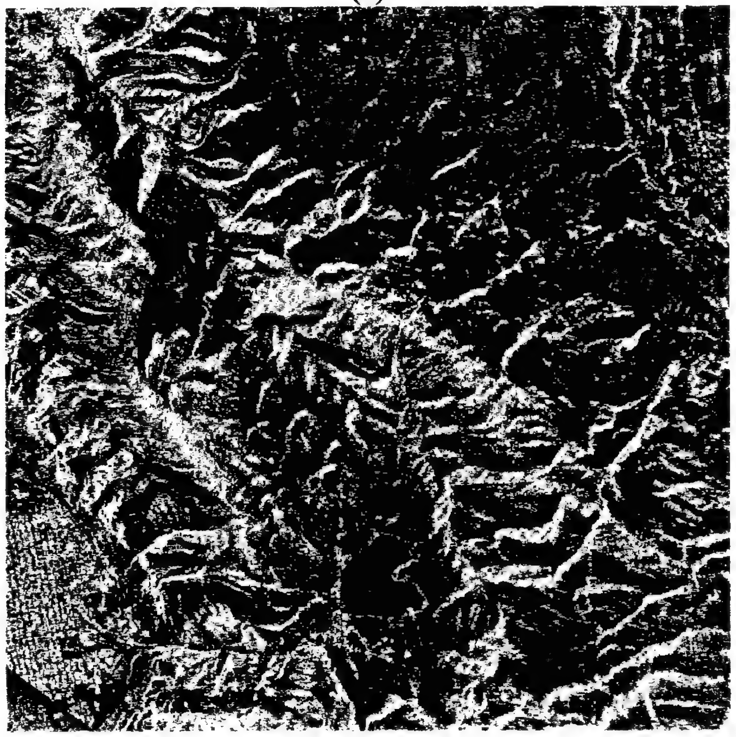


Fig. 6. (a) Landsat and (b) SAR source images of the Mantua, Utah, area.

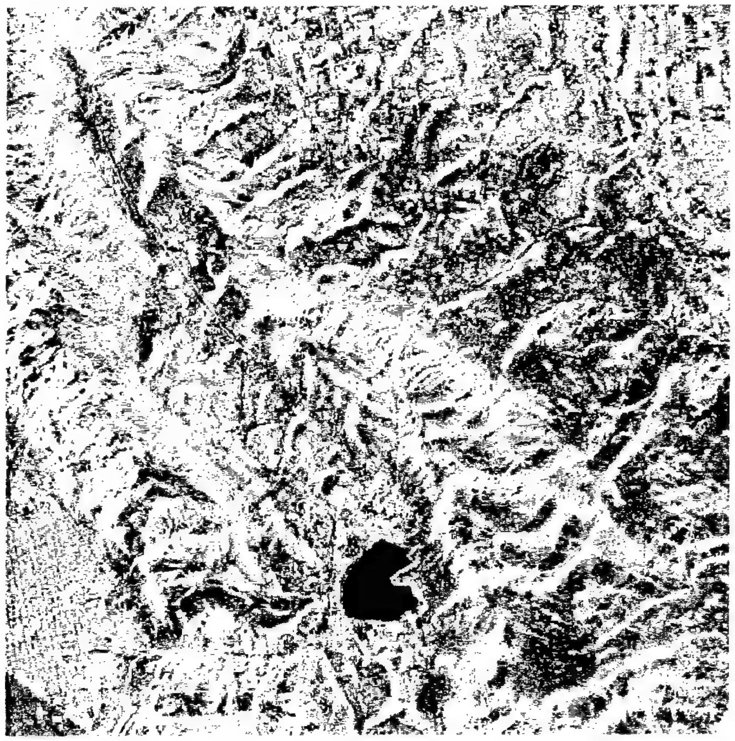


(a)

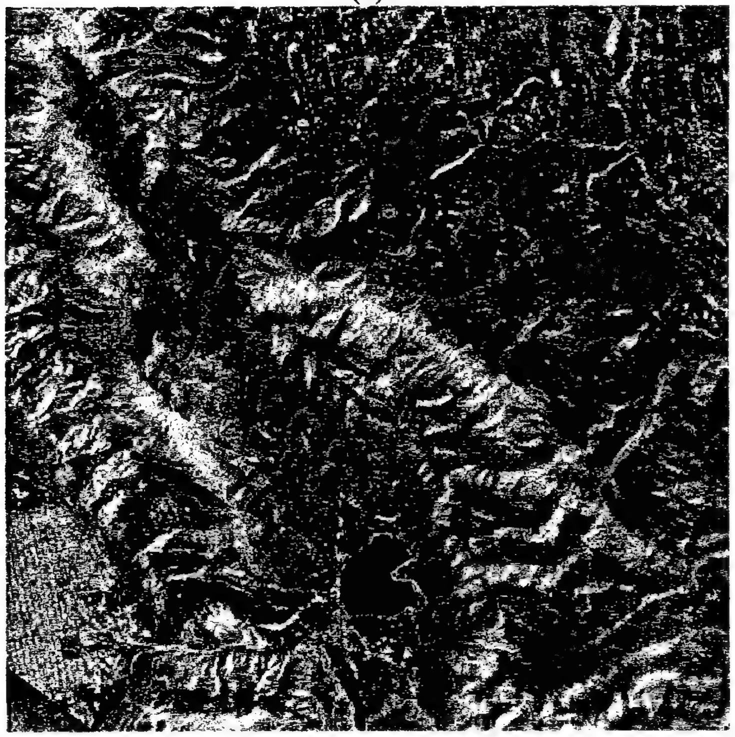


(b)

Fig. 7. Fusion of the Mantua, Utah, area source images, Figure 6 (a) and 6 (b), using a gradient pyramid and (a) selection fusion, (b) hybrid fusion.

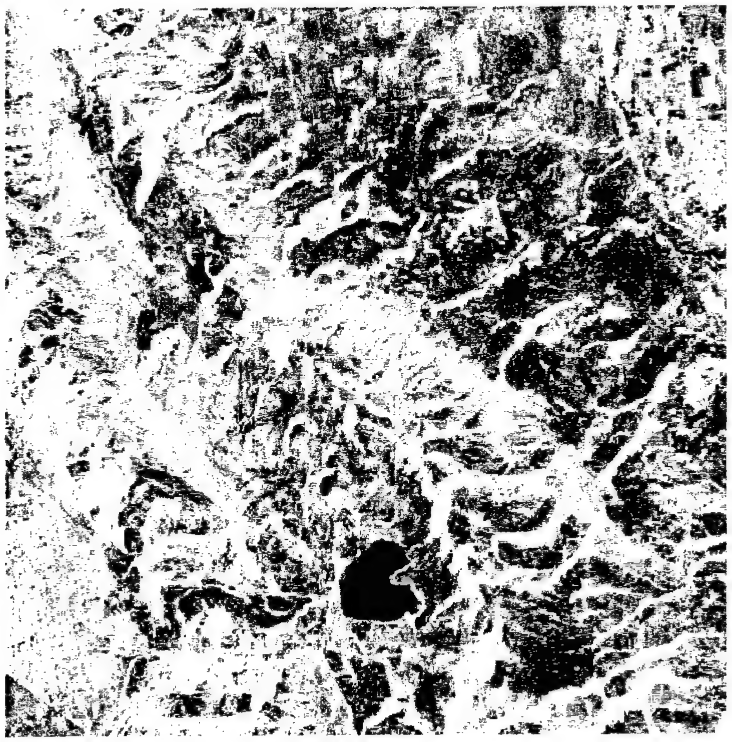


(a)



(b)

Fig. 8. Fusion of the Mantua, Utah, area source images, Figure 6 (a) and 6 (b), using a RoLP pyramid and (a) selection fusion, (b) hybrid fusion.


(a)

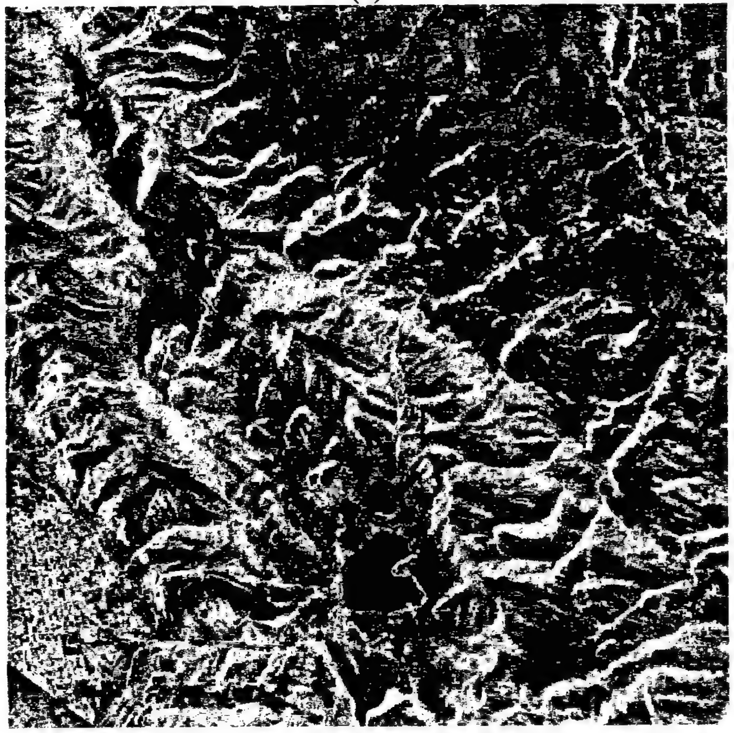

(b)

Fig. 9. Fusion of the Mantua, Utah, area source images, Figure 6 (a) and 6 (b), using a morphological pyramid and (a) selection fusion, (b) hybrid fusion.

visual results of the fusion techniques using morphological pyramids, Figure 9, show that the morphological pyramids do not work well for remote sensor imagery. The low quality of the composite images that used morphological filters is due to the small resolution of the visual elements; for example, roads are only one or two pixels wide. In this case, using a 2x2 or a 3x3 structuring element for the morphological filters does not allow reconstruction of such small detail. The image detail cannot be reconstructed with a resolution higher than that of the structuring element. Hence, the composite image does not contain the important, small details from the source images because the details are smaller than the structuring element. A smaller structuring element is not feasible because using a structuring element on the order of one pixel does not filter the image at all. Morphological filters work well for high-resolution images where image substructures are large in comparison with the structuring element. However, with these images the entire city is blurred.

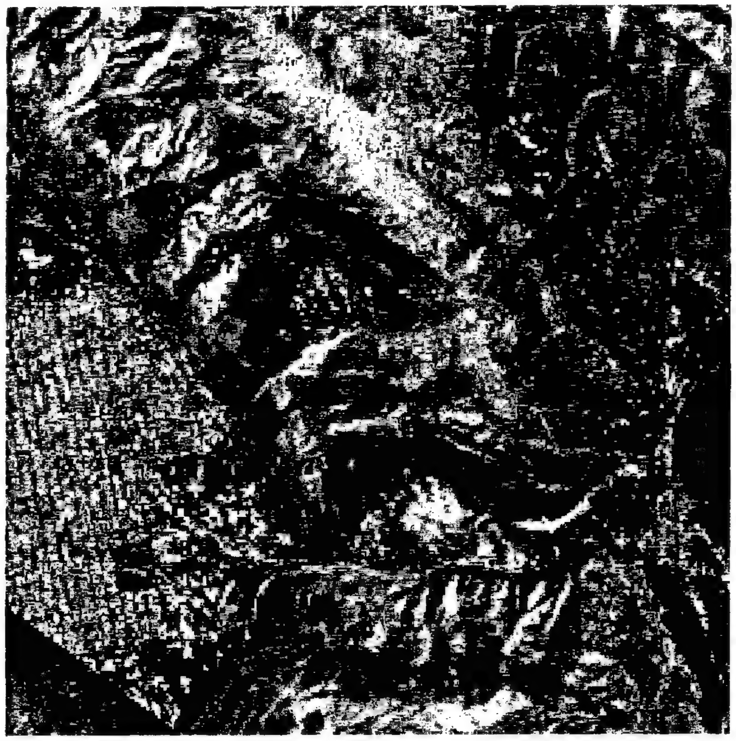
Results obtained using the selection technique for image fusion are unacceptable in this study because they select details from only one image or another. In the composite images, cities and mountains end up being represented by the SAR information, and agricultural areas are represented by the Landsat images. This is because Landsat images have the highest return values from agricultural areas, and SAR images have the highest return values in cities and mountains. Since the areas of saturated return correspond to the highest pixel values and the contrast selection technique is based upon ratios of pixel intensity, selection fusion leads to composite images that have the SAR information in city areas and Landsat information in the agricultural areas.

Figure 10 is an enlarged subsection of the two fused images that yielded the best result, namely, RoLP and gradient pyramids with hybrid fusion, Figures 7 (b) and 8 (b). Notice that the detail in the lower left-hand corner of the RoLP image appears blurry as compared with the same area in the gradient composite image.

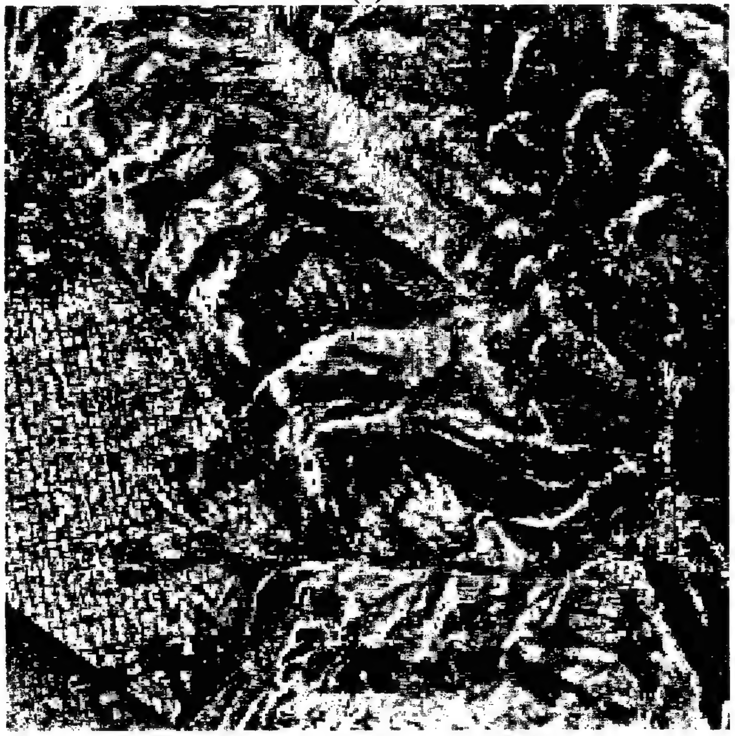
C. Fusing Hybrid SAR/Landsat Images

In the previous section, SAR and Landsat images were fused to obtain information about how the fusion functions work for remotely sensed imagery. Fusion applications, however, are more likely to use hybrid images. Hybrid images have the advantage of using multiple sensors to obtain single source images. The benefit of having the different bands of TM sensors and SAR sensors is that each band shows a particular feature of the surface. When these features are understood, source images can be formed to show specific information about a given surface area. In this section, the fusion techniques are applied to a sets of source images used to show the applicability of fusion of hybrid SAR and Landsat images. A list of the figures used in this section is given in Table IV.

For the first set of composite images, a source image designed to show urban, suburban, and agricultural areas is combined with another source image designed to show health of vegetation. For the first image, we want to use a band that reflects vegetation and a band that reflects anthropological structures; this will demonstrate the distinct difference between urban and agricultural areas. Landsat band 2 is in the visible



(a)



(b)

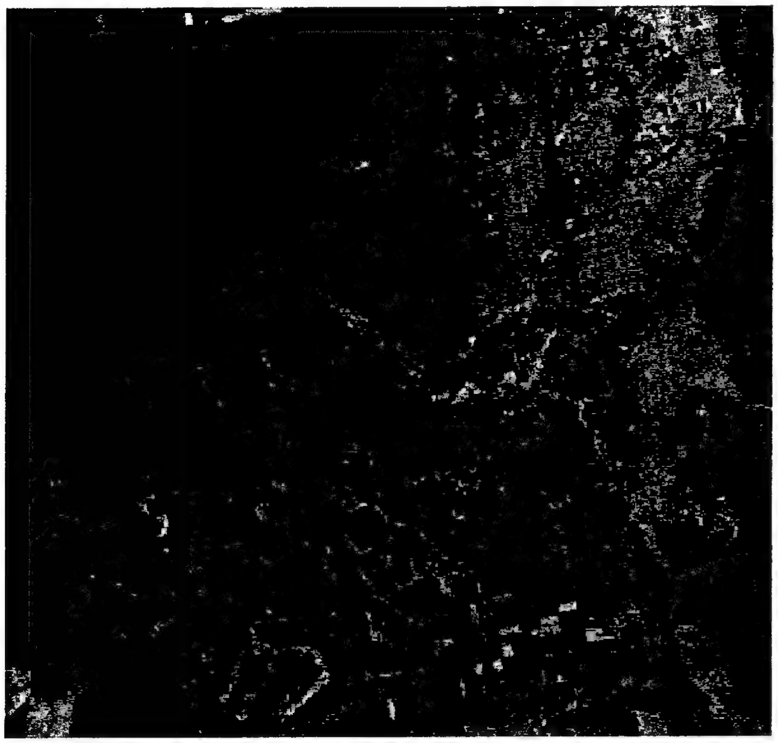
Fig. 10. (a) Gradient pyramid with hybrid fusion enlarged for detail. (b) RoLP pyramid with hybrid fusion enlarged for detail.

TABLE IV
LIST OF FIGURES IN CHAPTER V, SECTION C

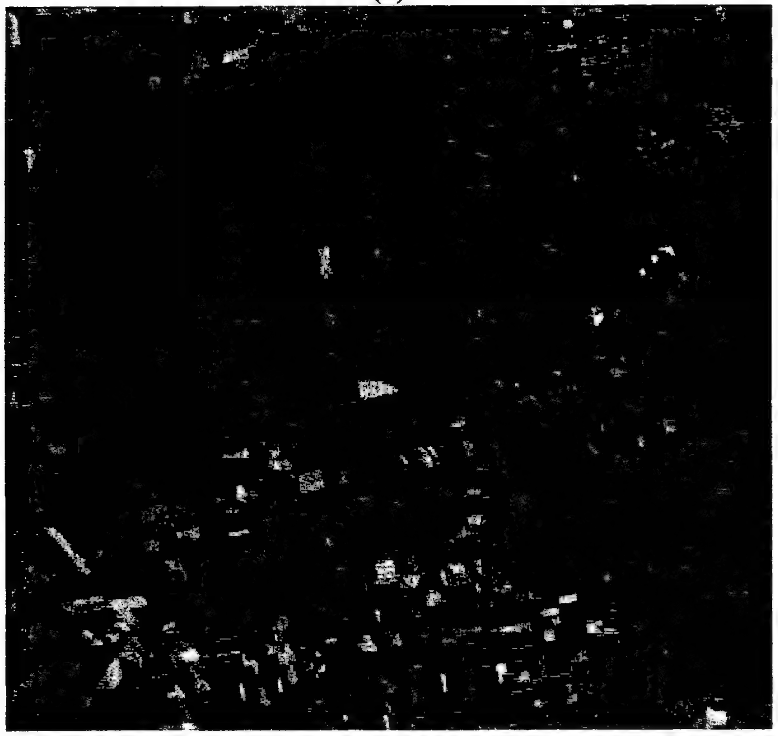
Figure	Area	Description
11 (a)	Logan	SAR band L and Landsat bands 4 and 2.
11 (b)	Logan	Landsat bands 3, 4, and 2.
12 (a)	Logan	Hybrid fusion of 11 (a) and 11 (b) using a four level Gradient pyramid with a Gaussian kernel of a = 0.4.
12 (b)	Logan	Hybrid fusion of 11 (a) and 11 (b) using a four level RoLP pyramid.

spectrum and returns a peak value for vegetation. Landsat band 4 is in the near infrared spectrum and shows healthy vegetation and land/water interfaces. Both SAR bands C and L reflect well from artificial structures and would work well for this image; however, band L has a higher return from artificial structures than band C, and band C reflects from vegetation. For the first image, we use Landsat bands 2 and 4 and SAR band L. The second source image is designed to show health of vegetation, so both Landsat bands 2 and 4 are again used. The third band of the second source image is one that has a low return in vegetated areas; Landsat band 3 is in the visible-light spectrum that corresponds to chlorophyll absorption.

Figure 11 shows the two hybrid source images. The first image emphasizes land use categories and land/water boundaries, and the second shows health of vegetation and land/water boundaries. Figure 12 shows the result of fusing the source images using RoLP and gradient pyramids with hybrid fusion. It is concluded that the fusion of the source images is successful because the information represented by Landsat bands 2 and



(a)



(b)

Fig. 11. (a) SAR band L and Landsat bands 4 and 2. (b) Landsat bands 3, 4, and 2.

(a)

(b)

Fig. 12. Fusion of source images in Figure 11 using a (a) Gradient pyramid with hybrid fusion. (b) RoLP pyramid with hybrid fusion.

4, which show health of vegetation and water/land boundaries, remains virtually unchanged, whereas the composite images make viewing the vegetation in the city easier without changing the ease of viewing the land use information. This is because the pixels represented by the color red are now the data represented by the fusion between the SAR L band and the Landsat band 3. This does not increase the amount of green present in the image; it only makes it easier for a human analyst to observe because of the way we perceive contrast. The reason for this is described by Weber's Law [9]. By decreasing the amount of red in the local area, it decreases the amount of contrast between the green and red pixels; hence, the vegetation (green) is more easily observed by a human analyst because the red is reduced. If we were to simply reduce the intensity of the red pixels, the contrast between green and red in the city would be still be easily observed; however, in areas where the red pixels are the only source, the contrast would also be reduced. The use of fusion allows varying the amount of change in contrast for a given area based on the correlation between the two source images. Figure 13 demonstrates the change in contrast by viewing enlarged subsections of Figures 11 (a), 12 (a), and 12 (b)

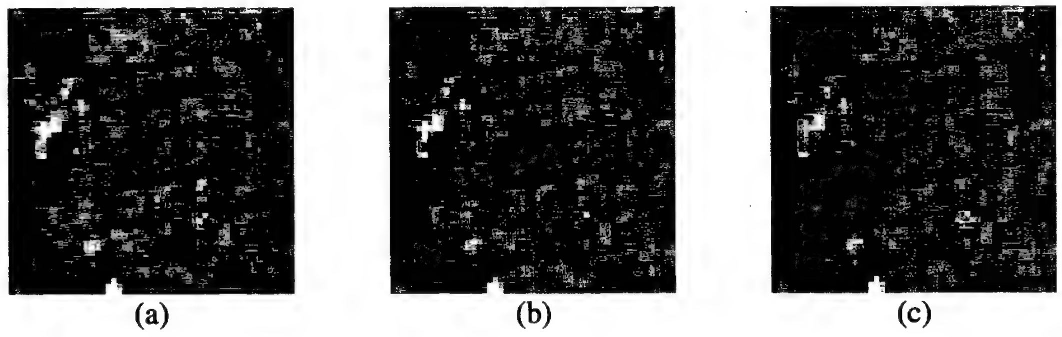


Fig. 13. Enlarged subsections of Figures 11 (a) and 12 to show the change in contrast.

respectively; the enlarged subsections in Figure 13 are: (a) the hybrid image, (b) the fused images using gradient pyramids, and (c) the fused images using RoLP pyramids.

To most observers it appears that Figure 13 (c) has the most green and Figure 13 (a) has the least with Figure 13 (b) somewhere in between. However, all three images have the same amount of green in them. The green pixel values did not change; only the red pixel values changed. By decreasing the intensity of the red pixel values, the green is more easily noticed; hence, people see more green in the images (b) and (c) of Figure 13. The same effect could be observed over small areas by simply decreasing the intensity of the red pixels in the image; however, fusion allows the amount of change in the pixel intensity to vary dependent upon the correlation of the two images. Fusion is beneficial because the change in the red pixel intensities is not constant over the whole image. The fusion algorithm decides how much and where to change the red pixel intensities.

D. Composite Image Appearance and Quality

The composite image appearance depends upon several factors. In this study, Landsat and SAR images were converted into bitmaps for fusion. This preserved the pixel data. In order for the REDUCE and EXPAND functions to work properly, the input image size needs to be equal to a power of two, plus one. For example, the two sizes used in this study were 257x257 and 513x513. This works because $257 = 2^8 + 1$ and $513 = 2^9 + 1$. If a source image is passed to the function that generates a pyramid, the image needs to have x and y sizes that are a power of two, plus one. If the dimensions of the source image do not meet this criterion, the image needs to be resized. When resizing occurs, pixel values must be interpolated; and the data used are no longer exact. Therefore, the data in the composite image are not exact.

Another consideration in the composite image appearance is the storage format used. If an image is stored as a JPEG file, the true pixel values are not saved; quantization is necessary for the compression of the file. Once again, because the data used in the source images are not exact, the data contained in the composite image are not exact. It is necessary for the user to decide what accuracy of pixel values is necessary for image analysis and take the necessary precautions when fusing the images.

When using multiresolution image fusion, the number of levels in the pyramids contributes to the quality of the composite image. For example, when the source images are decomposed into pyramids six levels deep, large image features will fuse better than if pyramids only two levels deep were used. The depth of a pyramid is an important parameter. If the depth is too deep, processing time is wasted. On the other hand, if the pyramid is not deep enough, the larger image subfeatures will not blend well. If it is known that the remote sensing imagery for a desired study has surface/subsurface features of a very small scale, the pyramid depth does not need to be as deep; a depth of two would work fine. In relation to the overall size of the source images used here, the features in the city are very small, while the features in the mountains are quite a bit larger. It is important to note that the pyramid depth must accommodate the fusion of the largest subfeature in the source images; this is what will determine the necessary pyramid depth. Figure 14 shows images created by fusing two images using gradient pyramids of different depths. Figure 15 shows the same thing using RoLP pyramids; the depths used range from one to six.

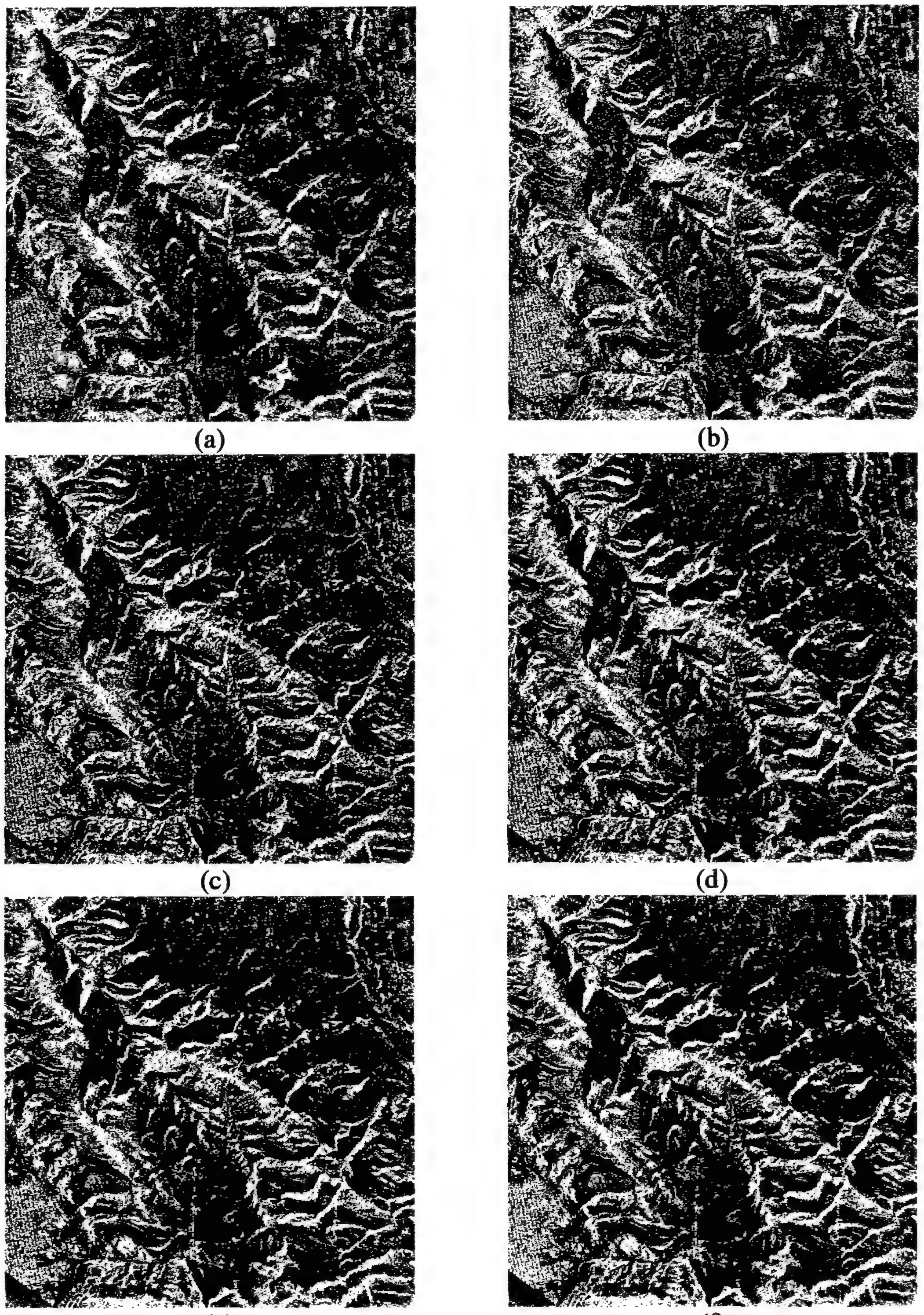


Fig. 14. Gradient pyramid hybrid fusion of the Mantua, Utah, area source images, Figure 9 (a) and 9 (b), using pyramids of varying depths (a) depth = 1 (b) depth = 2 (c) depth = 3 (d) depth = 4 (e) depth = 5 (f) depth = 6.

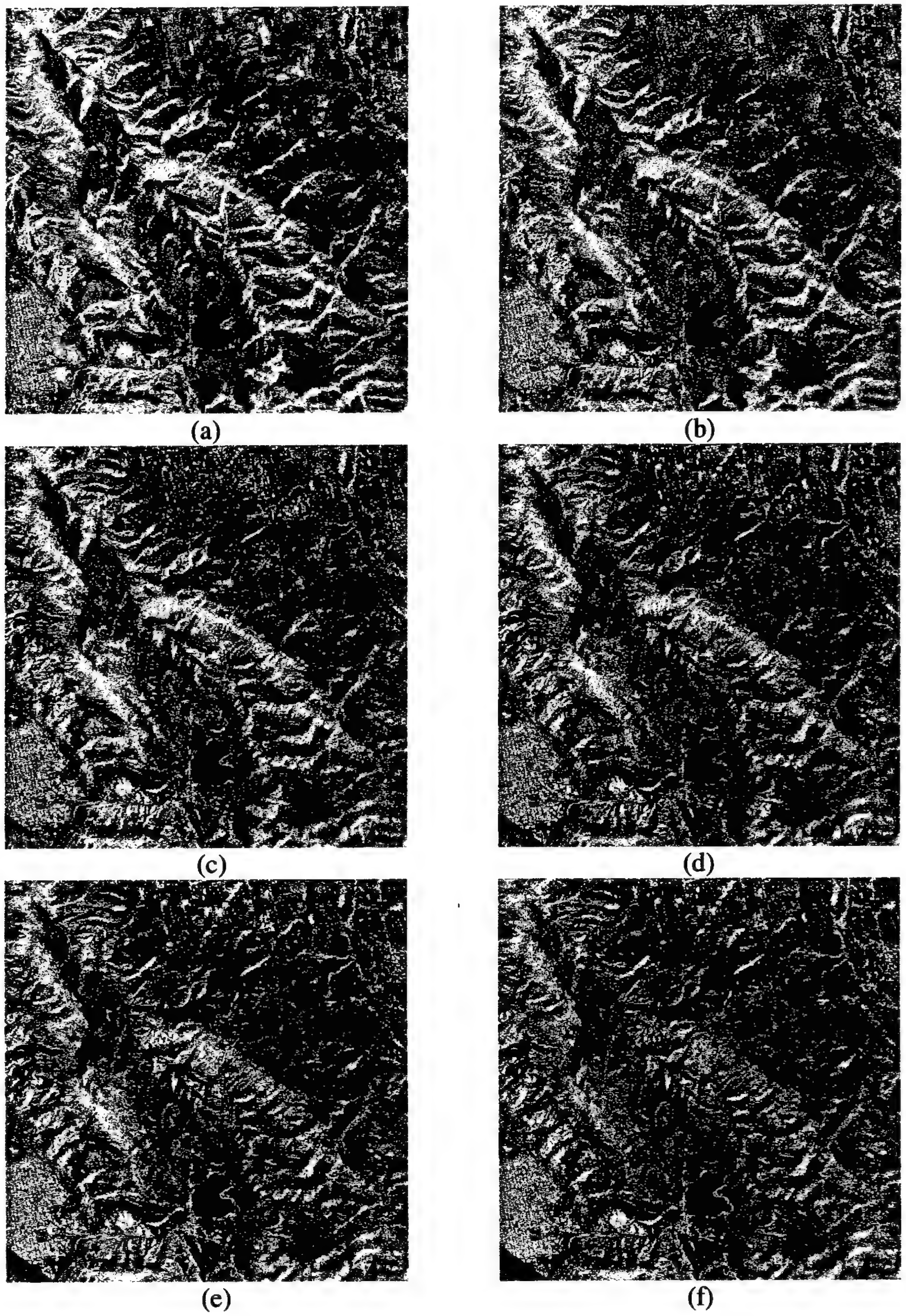


Fig. 15. RoLP pyramid hybrid fusion of the Mantua, Utah, area source images, Figure 9 (a) and 9 (b), using pyramids of varying depths (a) depth = 1 (b) depth = 2 (c) depth = 3 (d) depth = 4 (e) depth = 5 (f) depth = 6.

Comparing the images (in Figure 14 for gradient pyramids or Figure 15 for RoLP pyramids) with each other, it can be seen that, after a pyramid depth of two, the added pyramid levels do not visually add much detail to the composite image. The most noticeable change, upon visual inspection, is in the color of the mountains. To more clearly see the change that results from increasing the pyramid depth from level to level, Figure 16 shows the difference in gradient images, and Figure 17 shows the difference in RoLP images. (The difference of each pixel value has been multiplied by a factor of 4 to more clearly see the changes.)

By comparing the images in Figure 16 with the images in Figure 14 for gradient pyramids (or Figure 17 with Figure 15 for RoLP images), it is noticed that the difference from level to level is, indeed, in the mountainous regions of the image. The difference in the mountains was expected because the image features in the mountains are much larger than the image features in the city. By inspecting the resultant images, it can be observed that the detail in the composite images for the city areas did not change noticeably for any of the pyramid depths used after a depth of two. For any set of images, the optimum pyramid depth depends upon the size of the details considered for analysis. The larger the details in the source images, the deeper the pyramid depth needs to be for satisfactory fusion. If the pyramids were skipped all together (a pyramid of depth = 1), the composite image has a higher level of detail missing as opposed to using a depth of two.

This study also used a subjective measure of image quality as described by [24]. The quality of each image has been assigned a value between 0 and 9. An image with a quality measure (QM) of 1 would be able to locate only large terrain features such as rail yards, airstrips, or possibly very large aircraft at an airstrip. An image with a QM of 8,

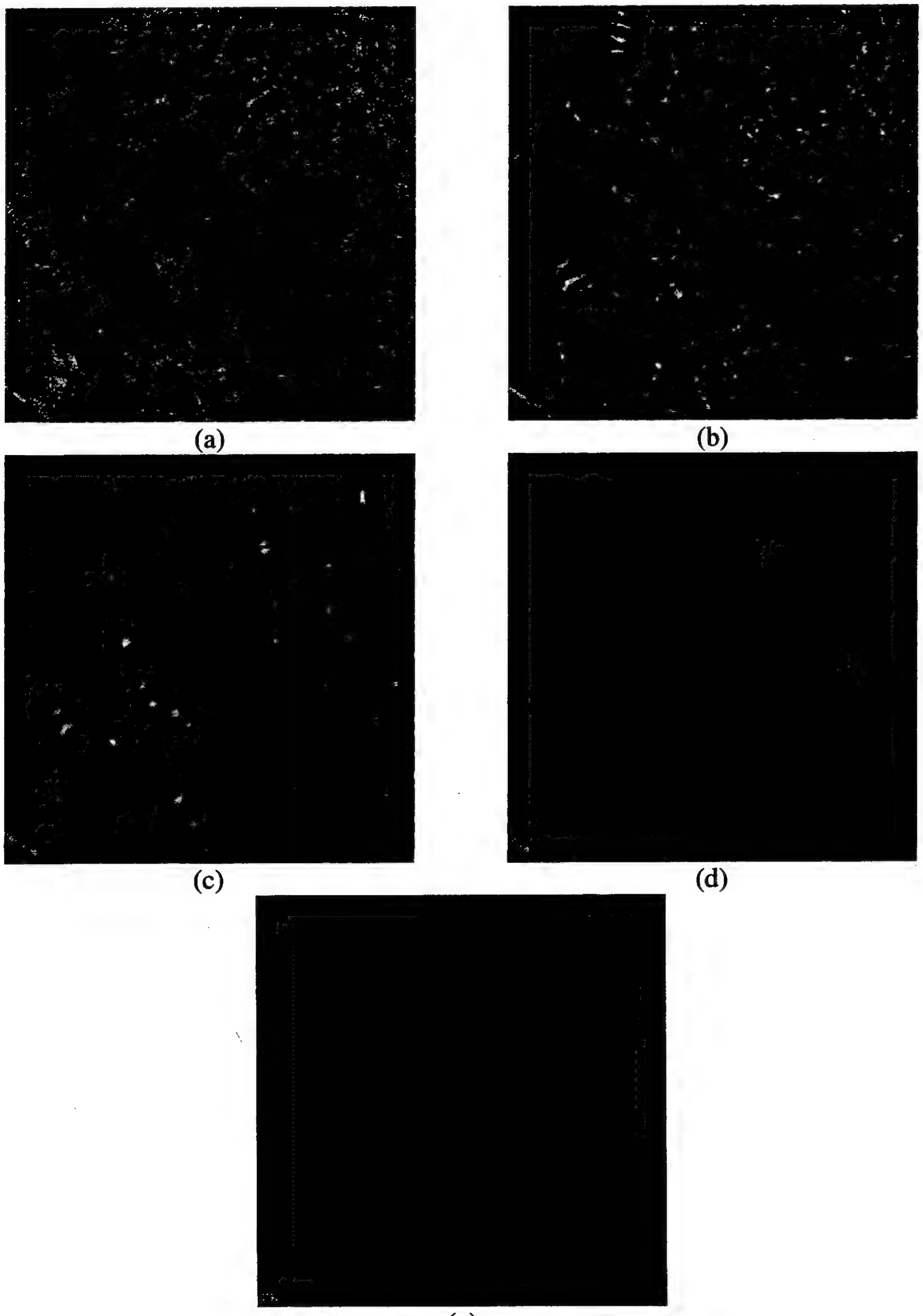

(a) (b) (c) (d) (e)

Fig. 16. Four times the difference between gradient composite images of successively increasing pyramid depths. (a) Fig. 14 (b) - Fig. 14 (a). (b) Fig. 14 (b) - Fig. 14 (c). (c) Fig. 14 (d) - Fig. 14 (c). (d) Fig. 14 (e) - Fig. 14 (d). (e) Fig. 14 (f) - Fig. 14 (e).

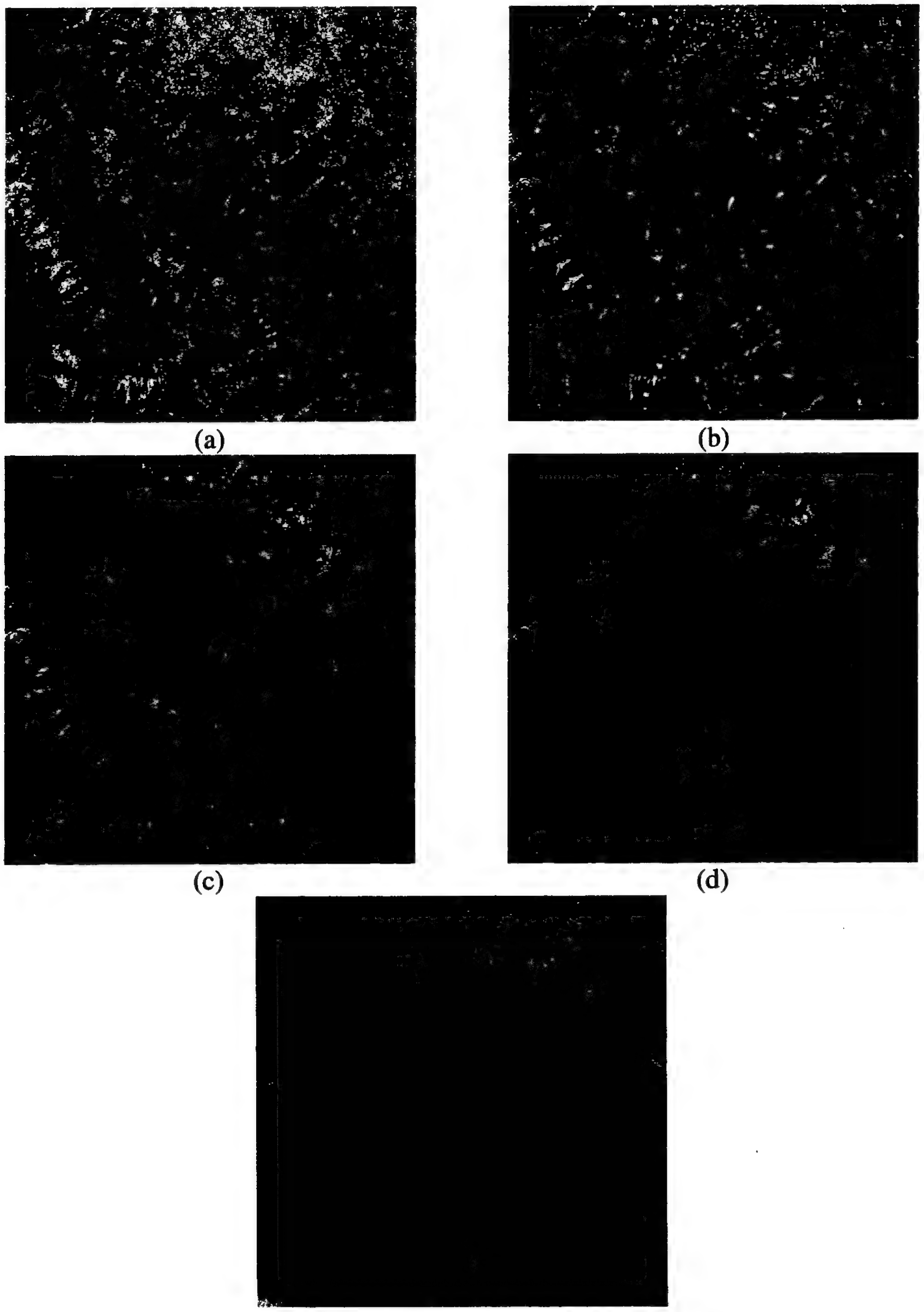

(a)
(b)
(c)
(d)
(e)

Fig. 17. Four times the difference between RoLP composite images of successively increasing pyramid depths. (a) Fig. 15 (b) - Fig. 15 (a). (b) Fig. 15 (b) - Fig. 15 (c). (c) Fig. 15 (d) - Fig. 15 (c). (d) Fig. 15 (e) - Fig. 15 (d). (e) Fig. 15 (f) - Fig. 15 (e).

on the other hand, would be able to detect types of aircraft at an airstrip or the number of boxcars at a rail yard. The image quality was obtained from the power spectral density of the image luminance. The image luminance is acquired by transforming from the RGB coordinate system to the XYZ coordinate system, where Y is the image luminance. (Appendix B contains the transform equations and the image QMs for the composite images in this study.) This quality measure indicated that image quality decreased as the pyramid depth increased. It also indicated that the composite morphological images had higher quality than the RoLP and gradient composite images, which is not the case, as can be visually verified. It is, therefore, concluded that the quality measure as described in [24] does not work well for fused imagery of remotely sensed data.

CHAPTER VI

CONCLUSIONS

A. *Findings*

The findings from this study are verified by the fused images generated from SAR and TM data of the same terrestrial scenes. It has been shown that there is a more effective way to view composite SAR and Landsat images than by simply viewing three bands at one time. The cost of the more effective composite image is computation time. Table V is a summary of processor time for the various fusion techniques that were studied.

The fastest fusion techniques use morphological filters; however, as the composite images indicate, morphological filters do not work well for these types of remotely sensed data because of the small-scale of image subfeatures in comparison with the structuring element.

TABLE V
COMPUTATION TIMES FOR FUSION TECHNIQUES

Technique	Size (Pixels)	Time (Seconds)
Morphological-Selection	513x513	116.2
RoLP-Selection	513x513	130.1
Morphological-Hybrid	513x513	260.2
RoLP-Hybrid	513x513	631.2
Gradient-Selection	513x513	1142.6
Gradient-Hybrid	513x513	1823.4

Computational time on a Pentium 266 MHz processor with 64 MB of RAM.
For an image size of 257x257, all computational times should be quartered.

Fusion techniques using RoLP pyramids are the second fastest. The composite images formed from RoLP techniques are beneficial in that they present the data from both source images in the composite scene better than do the simple hybrid images used in previous studies. Compared with techniques that use gradient pyramids, however, the composite images from RoLP pyramids tend to appear blurry. This appearance of being blurry, however, is an artifact of the reduced contrast in the composite image. The quality remains approximately constant between the gradient and RoLP composite images (given equal pyramid depths).

Gradient pyramids yielded composite images in approximately equal quality to, or slightly better than, the RoLP pyramids (gradient composites do not appear as blurry); however, the computation time for gradient pyramids is about four times that of the RoLP pyramids. The gradient pyramid is actually a combination of four pyramids. When exact analysis of imagery is necessary, the gradient pyramid may be the best choice; however, for most applications, the RoLP pyramid with hybrid fusion will work just as well.

It was found in this study that the selection approach to image fusion is undesirable. Because the selection technique chooses only one band or another to represent, it does not fuse them. Although the selection technique is almost twice as fast as the hybrid selection and averaging technique, the hybrid approach yields a composite image with better detail. A result of this study, therefore, is the suggestion that, for general fusion applications, a RoLP pyramid with hybrid selection and averaging fusion should be used. If small details are a concern and time is not a constraint, the gradient pyramid with hybrid fusion may yield a slightly better result.

This study has demonstrated that the optimum pyramid depth depends upon the largest important subfeature in the image. If the features of concern are small in detail, like the cities in the case of satellite data, a pyramid depth of two will produce the same result as a pyramid depth of six. On the other hand, if the features in the image are large, like mountains, a much deeper pyramid is necessary.

A simple rule to follow would be to use a pyramid depth of two if the image features desired for fusion are on the order of one to two pixels. For most applications, a pyramid depth of three would work fine. If the image features are very large, on the order of hundreds of pixels, a pyramid depth of six would be appropriate. Rarely would a pyramid depth of greater than five or six be needed. Based upon the results from figures 16 and 17, a pyramid depth of at least two should be used in all fusion applications because the amount of detail added when changing from a pyramid of depth one to a pyramid of depth two is sufficiently large.

B. Recommendations

Future research related to the topic of this study is needed in several areas. First, an optimum alpha value (the threshold that is used to regulate averaging and selection of composite pixel values) for the match and saliency measure could be established through research. New approaches need to be developed to speed up the gradient filters, thus greatly benefiting the software already developed. Also, blind source separation [25] can be explored as a means to image fusion. Another area of research is to find more suitable structuring elements for morphological filters. Structuring elements that would make possible the retention of small-scale image features in the composite image are needed.

Another area of a study could be the comparison between using the wavelet transform or using pyramids for multiresolution representation of images. Another type of pyramid/fusion combination that could be tried is a ratio between levels of a morphologically filtered pyramid with both fusion types. The final suggestion I have for further research is to develop another way to objectively measure image quality without knowing the image source. Such a metric would be a valuable tool for the field of image processing and analysis in general. Further research in these areas could greatly benefit the field of image processing as well as the analysis of remote sensing imagery.

REFERENCES

[1] P. L. Vornberger and R. A. Bindschadler, "Multi-spectral analysis of ice sheets using co-registered SAR and TM imagery," *Int. J. Remote Sensing*, vol. 13, pp. 637-645, Mar. 1992.

[2] R. Bindschalder and P. Vornberger, "Interpretation of SAR imagery of the Greenland ice sheet using coregistered TM imagery," *Remote Sens. Environ.,* vol. 42, pp. 167-175, Dec. 1992.

[3] B. N. Haack and E. T. Slonecker, "Merged spaceborne radar and thematic mapper digital data for locating villages in Sudan," *Photogrammetric Engineering & Remote Sensing*, vol. 60, pp. 1253-1257, Oct. 1994

[4] B. Chapman, "Using RADAR data: science and applications," 1998, available on the World Wide Web at URL: http://southport.jpl.nasa.gov/scienceapps/.

[5] D. Oliver, "*Image processing for integration of Landsat Thematic Mapper imagery with SIR-C synthetic aperture radar imagery.*" M.S. thesis, Utah State University, Logan, UT. 1996.

[6] Anonymous, "Satellite imaging exploring the earth from space: obtaining Landsat images," 1998, available on the World Wide Web at URL: http://members.aol.com /landsatcd/index.html.

[7] S. I. A. Rahman and K. J. Dalsted, "Using SIR-A and Landsat data to study the potential of the southwestern desert of Egypt for agricultural expansion," in *Proc. of the 23rd Int. Symp. on Remote Sensing of Environment*, 1990, pp. 949-958.

[8] T. A. Wilson, S. K. Rogers, and M. Kabrisky, "Perceptual based image fusion for hyperspectral data," 1994, available on the World Wide Web at URL: http://www.afit.mil/Schools/EN/ENG/LABS/PatternRec/publications/IEEE95/IEEE95.html

[9] A. K. Jain, *Fundementals of Digital Image Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1989.

[10] G. Huadong, L. Qizhong, and F. Xiuyin, "Detection of gold-bearing alteration zones using Landsat TM and airborne SAR in north Xinjiang region," in *Proc. of the 23rd Int. Symp. on Remote Sensing of Environment*, 1990, pp. 639-647.

[11] A. Toet, "Hierarchical image fusion," *Machine Vision and Applications*, vol. 3, no. 1, pp. 1-11, 1990.

[12] P. J. Burt and R. J. Kolczynski, "Enhanced image capture through fusion," *1993 IEEE 4th International Conf. on Computer Vision*, vol. 4, pp. 173-182, Feb. 1993.

[13] S. Marshall, G. K. Matsopoulos, and J.N.H Brunt, "Multiresolution morphological fusion of MR and CT images of the human brain," in *Mathematical Morphology and Its Applications to Image Processing*, J. Serra & P. Soille, Eds., Dordrecht, Netherlands: Kluwer Academic Publishers, 1994, pp. 317-324.

[14] P. J. Burt and E. H. Adelson, "The laplacian pyramid as a compact image code," *IEEE Trans. on Comm.*, vol. 31, no. 4, pp. 532-540, 1983.

[15] A. Toet., "A morphological pyramid image decomposition," *Pattern Recognition Letters*, vol. 9, no. 4, pp. 255-261, 1989.

[16] A. Toet, L. J. van Ruyven, and J. M. Valeton, "Merging thermal and visual images by a contrast pyramid," *Optical Engineering*, vol. 28, no. 7, pp. 789-792, 1989.

[17] A. Toet, "Multiscale contrast enhancement with applications to image fusion," *Optical Engineering*, vol. 31, no. 5, pp. 1026-1031, 1992.

[18] W. Pfaff, *"Multispectral change detection and interpretation using selective principal components and the tasseled cap transformation.*" M.S. thesis, Utah State University, Logan, UT. 1998.

[19] W. L. Wolf, and G. J. Zissis, *The Infrared Handbook.* Ann Arbor, MI: Environmental Research Institute of Michigan, 1993.

[20] Anonymous, "Landsat Thematic Mapper: data characteristics," 1998, available on the World Wide Web at URL: http://edcwww.cr.usgs.gov/glis/hyper/guide /landsat_tm.

[21] B. N. Koopmans and G. Forero R., "Airborne SAR and Landsat MSS as complementary information source for geological hazard mapping," *ISPRS J. of Photogrammetry and Remote Sensing*, vol. 48, pp. 28-37, Dec. 1993.

[22] R. M. Haralick, R. Stanley, and X. Zhuang, "Image analysis using mathematical morphology," *IEEE Trans. On Pattern Analysis and Machine Intellegence*, vol. 9, no. 1, pp. 532-549, 1987.

[23] R. M. Haralick, X. Zhuang, C. Lin, and J. S. J. Lee, "The digital morphological sampling theorem," *IEEE Trans. on Acoustics, Speech, and Signal Processing*, vol. 37, no. 12, pp. 2067-2089, 1989.

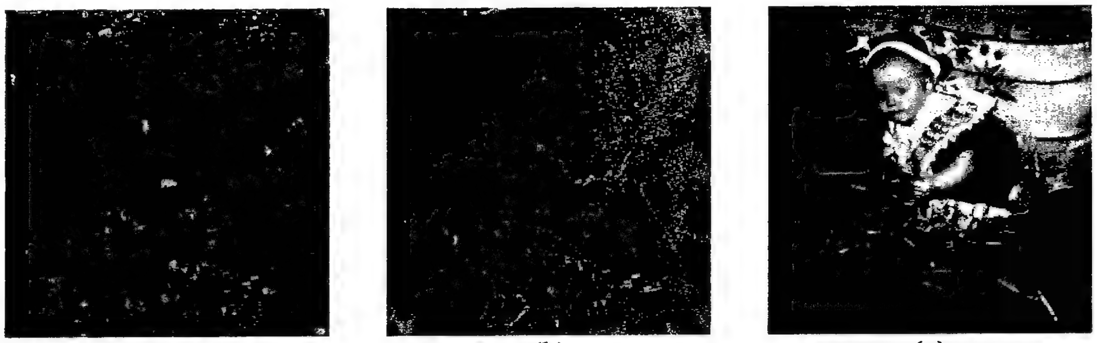
[24] N. B. Nill, and B. H. Bouzas, "Objective image quality measure derived from digital image power spectra," *Optical Engineering*, vol. 31, no. 4, pp. 813-825, 1992.

[25] A. J. Bell, and T. J. Sejnowski, “An information-maximization approach to blind source separation and blind deconvolution,” *Neural Comp.*, vol. 7, pp. 1129-1159, Feb. 1995.

APPENDICES

Appendix A: Testing of Pyramids and Fusion Code

Knowing that, theoretically, the RoLP pyramid should have an exact reconstruction and that morphological and gradient pyramids should have approximate reconstructions, the following tests were performed: First, a small test image of 17x17 pixels was created; then the pyramid routines were applied; and, at each stage of the image decomposition and reconstruction, the values were verified by hand calculations. The second test was to take three larger test images, shown in Figure 18, create a pyramid, and then recover the image by using the pyramid reconstruction techniques. Finally, the reconstructed image was subtracted from the original image; and the average and standard deviation of pixel differences was calculated. These values are given for the test images in Figure 18 in Tables VI, VII, and VIII. The second test was run with pyramids of depth = 6 and pyramids of depth = 4. For an exact reconstruction, the average pixel difference would be zero; and the standard deviation would also be zero. The testing showed that the RoLP pyramid was exact in most cases. The one exception was for a six level pyramid for image 18 (b) in the red band; in which case it was still very near zero. See Table VII. The morpological pyramids had the largest errors, and the errors were independent of pyramid depth. The errors in morphological pyramids are due to the fact that detail smaller than the structuring element cannot be exactly reconstructed. Gradient pyramids had a moderate margin of error for pyramid reconstruction; these errors are due to the approximation shown in equations (19) and (20). In general, the deeper the pyramid the larger the error in reconstructing the image.



(a) (b) (c)

Fig. 18. Test images used to verify pyramid code correctness.

TABLE VI
RESULTS FROM TESTING PYRAMID
RECONSTRUCTION CODE ON FIGURE 19 (a)

Pyramid Type	Pyramid Depth	Pixels Color	Average Error	Standard Deviation
Morph	6	Red	1.312237	2.981582
Morph	6	Green	2.045051	4.711925
Morph	6	Blue	0.810453	1.817184
RoLP	6	Red	0.000000	0.000000
RoLP	6	Green	0.000000	0.000000
RoLP	6	Blue	0.000000	0.000000
Gradient	6	Red	4.902764	2.869346
Gradient	6	Green	5.615083	5.245300
Gradient	6	Blue	4.639384	1.951420
Morph	4	Red	1.312237	2.981582
Morph	4	Green	2.045051	4.711925
Morph	4	Blue	0.810453	1.817184
RoLP	4	Red	0.000000	0.000000
RoLP	4	Green	0.000000	0.000000
RoLP	4	Blue	0.000000	0.000000
Gradient	4	Red	3.059642	1.689473
Gradient	4	Green	3.289907	2.882259
Gradient	4	Blue	2.969148	1.107655

TABLE VII
RESULTS FROM TESTING PYRAMID
RECONSTRUCTION CODE ON FIGURE 19 (b)

Pyramid Type	Pyramid Depth	Pixels Color	Average Error	Standard Deviation
Morph	6	Red	6.048186	16.756359
Morph	6	Green	2.045051	4.711925
Morph	6	Blue	0.810453	1.817184
RoLP	6	Red	0.000418	0.051939
RoLP	6	Green	0.000000	0.000000
RoLP	6	Blue	0.000000	0.000000
Gradient	6	Red	4.250710	4.461939
Gradient	6	Green	5.615083	5.245300
Gradient	6	Blue	4.639384	1.951420
Morph	4	Red	6.048186	16.756359
Morph	4	Green	2.045051	4.711925
Morph	4	Blue	0.810453	1.817184
RoLP	4	Red	0.000418	0.051939
RoLP	4	Green	0.000000	0.000000
RoLP	4	Blue	0.000000	0.000000
Gradient	4	Red	2.729921	2.905167
Gradient	4	Green	3.289907	2.882259
Gradient	4	Blue	2.969148	1.107655

TABLE VIII
RESULTS FROM TESTING PYRAMID
RECONSTRUCTION CODE ON FIGURE 19 (c)

Pyramid Type	Pyramid Depth	Pixels Color	Average Error	Standard Deviation
Morph	6	Red	1.887326	5.636188
Morph	6	Green	1.660585	4.812446
Morph	6	Blue	1.723327	4.875687
RoLP	6	Red	0.000000	0.000000
RoLP	6	Green	0.000000	0.000000
RoLP	6	Blue	0.000000	0.000000
Gradient	6	Red	5.413866	6.271135
Gradient	6	Green	4.584484	3.704715
Gradient	6	Blue	4.457820	3.544401
Morph	4	Red	1.887326	5.636188
Morph	4	Green	1.660585	4.812446
Morph	4	Blue	1.723327	4.875687
RoLP	4	Red	0.000000	0.000000
RoLP	4	Green	0.000000	0.000000
RoLP	4	Blue	0.000000	0.000000
Gradient	4	Red	3.306019	3.456579
Gradient	4	Green	3.044988	2.178596
Gradient	4	Blue	3.016866	2.076221

Appendix B: Image Quality Measures

The image quality measures used in this study were calculated from the power spectrum of the image as described in [24]. The power spectrum was taken on the image luminance, which was calculated by transforming from the RGB coordinates to the XYZ coordinates through the following transform:

$$X = 0.490*R + 0.310*G + 0.200*B;$$
$$Y = 0.177*R + 0.813*G + 0.011*B;$$
$$Z = 0.000*R + 0.010*G + 0.990*B;$$

where Y is the image luminance.

The quality measure was found as follows:

$$QM = 1.93*\log(IQM) + 8.77,$$

where

$$IQM = \frac{1}{M^2} \sum_{\theta=180^o}^{180^o} \sum_{\rho=0.01}^{0.5} S(\theta_1) W(\rho) A^2(T\rho) P(\rho,\theta) \text{ [24].}$$

Here, $P(\rho,\theta)$ is the Discrete Fourier Transform or the Fast Fourier Transform. $S(\theta_1)$ is the scale of the image at an angle of θ_1 from the sensor. $W(\rho)$ is the classic Wiener filter as described in [24]; and $A(T\rho)$ is a model of the human visual system (HVS), also described in [24].

The quality measures (QMs) were taken by dividing images into quadrants of size 128x128. An image of original size 513x513 would be divided into quadrants and then subdivided again into subquadrants. Quadrants start with number one in the upper right-hand corner of the image and then proceed counterclockwise to the lower right-hand corner, which is quadrant four. Table IX and X list the QMs for the composite images created in the study. The QMs from this study indicate that a quantitative measure of image quality that models the HVS exactly still needs to be developed. This quality measure did, however, provide a means to measure small differences in image quality and trends in changes based upon varying inputs such as pyramid depth.

TABLE IX
QUALITY MEASURES FOR COMPOSITE IMAGES IN FIGURES 7 THRU 12

Figure	Quadrant	S_Q1	S_Q2	S_Q3	S_Q4
7 (a)	1	5.743940	6.299867	5.955395	5.798092
7 (a)	2	5.774915	6.010545	6.144174	6.240824
7 (a)	3	6.031102	6.109628	5.829595	6.149844
7 (a)	4	5.537493	4.949940	5.178320	6.203316
7 (b)	1	5.731352	5.888458	5.774751	5.797175
7 (b)	2	5.455630	5.956551	5.839167	6.143242
7 (b)	3	5.871299	5.776086	6.154200	6.210767
7 (b)	4	5.686476	5.879713	5.157750	6.044588
8 (a)	1	6.216465	6.068850	6.105093	5.860565
8 (a)	2	2.987027	5.877355	5.704545	6.401773
8 (a)	3	6.054478	6.100063	6.158357	6.205588
8 (a)	4	3.824745	4.298860	5.471602	6.395981
8 (b)	1	5.683046	6.041419	5.824888	5.818359
8 (b)	2	5.043132	5.987381	5.779756	6.235353
8 (b)	3	5.879769	5.643103	6.231497	6.232010
8 (b)	4	5.597918	5.909996	5.156698	6.147477
9 (a)	1	5.542208	6.417702	5.934047	5.910607
9 (a)	2	6.183458	5.925222	4.363320	6.441932
9 (a)	3	6.170932	6.095793	6.064970	6.194372
9 (a)	4	4.960927	5.389463	5.162599	6.263077
9 (b)	1	5.757378	6.148040	5.833893	5.915841
9 (b)	2	6.171905	6.070212	5.708892	6.401994
9 (b)	3	5.996606	5.932110	6.246206	6.177732
9 (b)	4	4.377702	5.238811	5.299767	6.259186
11 (a)	1	4.182329	6.433823	6.187418	6.323740
11 (a)	2	5.828491	5.019885	5.546538	5.774281
11 (a)	3	5.602613	6.041763	5.954587	4.099891
11 (a)	4	6.281577	6.463826	5.614375	5.843056
11 (b)	1	4.182329	6.433823	6.187418	6.323740
11 (b)	2	5.828491	5.019885	5.546538	5.774281
11 (b)	3	5.602613	6.041763	5.954587	4.099891
11 (b)	4	6.281577	6.463826	5.614375	5.843056
12 (a)	1	4.971820	6.396582	6.191291	6.352974
12 (a)	2	5.787445	4.995888	5.494674	5.742789
12 (a)	3	5.592110	5.999828	5.919660	4.059719
12 (a)	4	6.216224	6.469916	5.645947	5.817126
12 (b)	1	4.170645	6.430707	6.190845	6.319584
12 (b)	2	5.826273	5.018929	5.545628	5.773333
12 (b)	3	5.604957	6.040998	5.953582	4.104306
12 (b)	4	6.282585	6.463218	5.615763	5.842026

Here S_Q1 stands for subquadrant 1, S_Q2 stands for subquadrant 2, S_Q3 stands for subquadrant 3, and S_Q4 stands for subquadrant 4.

TABLE X
QUALITY MEASURES FOR COMPOSITE IMAGES IN FIGURES 14 AND 15

Figure	Quadrant	S_Q1	S_Q2	S_Q3	S_Q4
14 (a)	1	5.907000	6.105254	5.954466	5.892723
14 (a)	2	6.134328	6.087080	6.248683	6.403800
14 (a)	3	6.060922	5.777319	6.310244	6.218068
14 (a)	4	5.394370	5.815682	5.238525	6.211088
14 (b)	1	5.909584	6.089230	5.919499	5.878862
14 (b)	2	6.041452	6.063083	6.129162	6.358926
14 (b)	3	6.009208	5.737093	6.263228	6.211189
14 (b)	4	5.493500	5.796942	5.213909	6.161926
14 (c)	1	5.860167	6.027700	5.870211	5.852189
14 (c)	2	5.782127	6.023773	5.988911	6.284763
14 (c)	3	5.948137	5.751993	6.220279	6.218034
14 (c)	4	5.596609	5.785432	5.199620	6.111913
14 (d)	1	5.731352	5.888458	5.774751	5.797175
14 (d)	2	5.455630	5.956551	5.839167	6.143242
14 (d)	3	5.871299	5.776086	6.154200	6.210767
14 (d)	4	5.686476	5.879713	5.157750	6.044588
14 (e)	1	5.449706	5.570606	5.653515	5.714482
14 (e)	2	5.736781	5.890406	5.472869	5.945735
14 (e)	3	5.797911	5.752635	6.020828	6.168142
14 (e)	4	5.714855	6.015885	5.034786	5.996046
14 (f)	1	5.155452	5.912589	5.555240	5.590639
14 (f)	2	6.083034	5.791876	4.703432	5.677327
14 (f)	3	5.674115	5.736203	5.816588	6.045265
14 (f)	4	5.674343	6.074334	4.679347	5.855797
15 (a)	1	5.907000	6.105254	5.954466	5.892723
15 (a)	2	6.134328	6.087080	6.248683	6.403800
15 (a)	3	6.060922	5.777319	6.310244	6.218068
15 (a)	4	5.394370	5.815682	5.238525	6.211088
15 (b)	1	5.894845	6.126315	5.929002	5.887159
15 (b)	2	5.856588	6.067307	6.129728	6.374213
15 (b)	3	6.004285	5.757083	6.264523	6.222906
15 (b)	4	5.463844	5.787212	5.208783	6.179084
15 (c)	1	5.840663	6.041954	5.889577	5.861843
15 (c)	2	5.188877	6.028841	5.963924	6.309322
15 (c)	3	5.939251	5.734792	6.252913	6.221472
15 (c)	4	5.520334	5.778367	5.183959	6.152245
15 (d)	1	5.683046	6.041419	5.824888	5.818359
15 (d)	2	5.043132	5.987381	5.779756	6.235353
15 (d)	3	5.879769	5.643103	6.231497	6.232010
15 (d)	4	5.597918	5.909996	5.156698	6.147477
15 (e)	1	5.313802	5.692202	5.792126	5.804777
15 (e)	2	5.652391	5.971659	5.431761	6.216579
15 (e)	3	5.864790	5.557136	6.161402	6.245972
15 (e)	4	5.673966	6.072573	5.161863	6.130646
15 (f)	1	4.772618	5.091612	5.772559	5.813745
15 (f)	2	6.126851	5.980894	4.989271	6.207485
15 (f)	3	5.839331	5.599754	6.058500	6.202442
15 (f)	4	5.702899	6.123158	5.102581	6.125611

Here S_Q1 stands for subquadrant 1, S_Q2 stands for subquadrant 2, S_Q3 stands for subquadrant 3, and S_Q4 stands for subquadrant 4.

Appendix C: Source Code

```
/*********************************** fuse_n.cpp ********************************/
/*   This file contains the code for fusing the images.  It contains both       */
/*   the fusion functions for selection and hybrid selection and averaging.     */
/*   It also contains the six fusion functions used to fuse RoLP, gradient,     */
/*   and morphological pyramids with the two fusion techniques.                 */
/********************************************************************************/
#ifndef FUSE
#define FUSE

#include "thesis.h"
#include "files.cpp"
#include "convolve_n.cpp"
#include "gradient_n.cpp"
#include "morph_n.cpp"
#include "rolp_n.cpp"


/********************************************************************************/
/*                        Image Fusion Techniques                               */
/********************************************************************************/
/* Fuses two images selecting between the two source pixels the pixel with
   the highest intensity; also called contrast fusion.  */
image * selection(image *i1, image *i2)
{
  image *newpic;
  int xnew, ynew, x, y, index;
  if(i1 == NULL || i2 == NULL) return(NULL);
  if(i1->xsize == i2->xsize && i1->ysize == i2->ysize)  {
    xnew = i1->xsize;
    ynew = i2->ysize;
    newpic = (image *)malloc(sizeof(image));
    newpic->pixel = (Pixel *)malloc(xnew*ynew*sizeof(Pixel));
    for(x = 0; x < xnew; x++)  {
      for(y = 0; y < ynew; y++)  {
        index = y*xnew + x;
        if( (i1->pixel[index][0] - 1) > (i2->pixel[index][0] - 1))  {
          newpic->pixel[index][0] = i1->pixel[index][0];
        }
        else  {
          newpic->pixel[index][0] = i2->pixel[index][0];
        }
        if( (i1->pixel[index][1] - 1) > (i2->pixel[index][1] - 1))  {
          newpic->pixel[index][1] = i1->pixel[index][1];
        }
        else  {
          newpic->pixel[index][1] = i2->pixel[index][1];
        }
        if( (i1->pixel[index][2] - 1) > (i2->pixel[index][2] - 1))  {
          newpic->pixel[index][2] = i1->pixel[index][2];
        }
        else  {
          newpic->pixel[index][2] = i2->pixel[index][2];
        }
      }
    }
    newpic->xsize = xnew;
    newpic->ysize = ynew;
    return(newpic);
  }
  else return(NULL);
}

/* This function is used for the match and saliency measures for hybrid
   selection and averaging fusion.  This function returns the weighted
   average of the product of a 3-by-3 area surrounding the pixel in the
   (i, j) position for a given pixel color.  */
double sample_p(image * pic1, image * pic2, int i, int j, int color)
{
  //The images must be the same x and y dimentions
  int m, n, index, tx, ty;
  double tsum = 0.0;
  for(m = -2; m < 3; m++)  {
    for(n = -2; n < 3; n++)  {
      tx = i + m;
      if(tx < 0 || tx >= pic1->xsize || tx >= pic2->xsize)   continue;

      ty = j + n;
      if(ty < 0 || ty >= pic1->ysize || ty >= pic2->ysize)   continue;

      index = ty*pic1->xsize + tx;
      tsum += w_g(m,n) * pic1->pixel[index][color] * pic2->pixel[index][color];
    }
  }
  return(tsum);
}

/* The salience function returns a corresponding map the same dimentions of the
   image which is used to determine which pixel to use for selection, or which
   one has a higher weight for averaging.  */
image * salience(image *g)
{
  image *newpic;
  int x, y, index;

  newpic = (image *)malloc(sizeof(image));
  newpic->pixel = (Pixel *)malloc(g->xsize*g->ysize*sizeof(Pixel));
  newpic->xsize = g->xsize;
  newpic->ysize = g->ysize;
  for(y = 0; y < g->ysize; y++)  {
    for(x = 0; x < g->xsize; x++)  {
      index = y*g->xsize + x;
      newpic->pixel[index][0] = sample_p(g, g, x, y, 0);
      newpic->pixel[index][1] = sample_p(g, g, x, y, 1);
      newpic->pixel[index][2] = sample_p(g, g, x, y, 2);
    }
  }
  return(newpic);
}

/*  This function reuturns a measure of match to by which to compare the two
    images.  If the two images match, averaging is used, if they don't match
    within the weighted average is used where the one with the higher
    saliency has the higher weight.  */
image * match(image *a, image *b, image *sa, image *sb)
{
  image *newpic;
  int x, y, index;

  newpic = (image *)malloc(sizeof(image));
  newpic->pixel = (Pixel *)malloc(a->xsize*a->ysize*sizeof(Pixel));
  newpic->xsize = a->xsize;
  newpic->ysize = a->ysize;
  for(y = 0; y < a->ysize; y++)  {
    for(x = 0; x < a->xsize; x++)  {
      index = y*a->xsize + x;
      newpic->pixel[index][0] = abs(2*sample_p(a, b, x, y, 0)/
                                (sa->pixel[index][0] + sb->pixel[index][0]));
      newpic->pixel[index][1] = abs(2*sample_p(a, b, x, y, 1)/
                                (sa->pixel[index][1] + sb->pixel[index][1]));
      newpic->pixel[index][2] = abs(2*sample_p(a, b, x, y, 2)/
                                (sa->pixel[index][2] + sb->pixel[index][2]));
    }
  }
  return(newpic);
}

/* Fuses two images by using both selection and averaging.  If the two pixels
   correspond to the same type of subfeature averageing is used, otherwise
   selection is used.  Alpha is a measure of how close the two pixels must
   match to use selection or averaging.  */
image * hybrid_selection_average(image *a, image *b, double alpha)
{
  image *sa, *sb, *m, *newpic;
  double wmin, wmax, wa, wb;
  int x, y, index, c;

  if(a == NULL || b == NULL) return(NULL);
  sa = salience(a);
  sb = salience(b);
  m = match(a, b, sa, sb);
  newpic = (image *)malloc(sizeof(image));
  newpic->pixel = (Pixel *)malloc(a->xsize*a->ysize*sizeof(Pixel));
  newpic->xsize = a->xsize;
  newpic->ysize = a->ysize;
  for(y = 0; y < a->ysize; y++)  {
    for(x = 0; x < a->xsize; x++)  {
      index = y*a->xsize + x;
      for(c = 0; c < 3; c++)  {
        wmin = .5 - .5*(1 - m->pixel[index][c])/(1 - alpha);
        wmax = 1 - wmin;
        if(sa->pixel[index][c] > sb->pixel[index][c])  {
          wa = wmax; wb = wmin;
        }  else  {
          wa = wmin;  wb = wmax;
        }
        newpic->pixel[index][c] = wa*a->pixel[index][c] + wb*b->pixel[index][c];
      }
    }
  }

  return(newpic);
}

/*********************************************************************************/
/*                   Fuse Gradient Pyramid with hybrid fusion                    */
/*********************************************************************************/
/* This function fuses two gradient pyramids using hybrid fusion  */
image * fuse_gradient_hybrid_pyramid(Pyramid_Gradient *A, Pyramid_Gradient *B, double
alpha)
{
  Pyramid_Gradient *C;
  image *comp;

  C = (Pyramid_Gradient *)malloc(sizeof(Pyramid_Gradient));
  C->d4 = (Pyramid *)malloc(sizeof(Pyramid));
  C->d3 = (Pyramid *)malloc(sizeof(Pyramid));
  C->d2 = (Pyramid *)malloc(sizeof(Pyramid));
  C->d1 = (Pyramid *)malloc(sizeof(Pyramid));
  C->g = (Pyramid *)malloc(sizeof(Pyramid));

  printf("\nFusing Gradient 4...");
  C->d4->I5 = NULL;
  C->d4->I4 = hybrid_selection_average(A->d4->I4, B->d4->I4, alpha);
  C->d4->I3 = hybrid_selection_average(A->d4->I3, B->d4->I3, alpha);
  C->d4->I2 = hybrid_selection_average(A->d4->I2, B->d4->I2, alpha);
  C->d4->I1 = hybrid_selection_average(A->d4->I1, B->d4->I1, alpha);
  C->d4->I0 = hybrid_selection_average(A->d4->I0, B->d4->I0, alpha);
  printf("...Done\n");

  printf("\nFusing Gradient 3...");
  C->d3->I5 = NULL;
  C->d3->I4 = hybrid_selection_average(A->d3->I4, B->d3->I4, alpha);
  C->d3->I3 = hybrid_selection_average(A->d3->I3, B->d3->I3, alpha);
  C->d3->I2 = hybrid_selection_average(A->d3->I2, B->d3->I2, alpha);
  C->d3->I1 = hybrid_selection_average(A->d3->I1, B->d3->I1, alpha);
  C->d3->I0 = hybrid_selection_average(A->d3->I0, B->d3->I0, alpha);
  printf("...Done\n");
  printf("\nFusing Gradient 2...");
  C->d2->I5 = NULL;
  C->d2->I4 = hybrid_selection_average(A->d2->I4, B->d2->I4, alpha);
  C->d2->I3 = hybrid_selection_average(A->d2->I3, B->d2->I3, alpha);
  C->d2->I2 = hybrid_selection_average(A->d2->I2, B->d2->I2, alpha);
  C->d2->I1 = hybrid_selection_average(A->d2->I1, B->d2->I1, alpha);
  C->d2->I0 = hybrid_selection_average(A->d2->I0, B->d2->I0, alpha);
  printf("...Done\n");

  printf("\nFusing Gradient 1...");
  C->d1->I5 = NULL;
  C->d1->I4 = hybrid_selection_average(A->d1->I4, B->d1->I4, alpha);
  C->d1->I3 = hybrid_selection_average(A->d1->I3, B->d1->I3, alpha);
  C->d1->I2 = hybrid_selection_average(A->d1->I2, B->d1->I2, alpha);
  C->d1->I1 = hybrid_selection_average(A->d1->I1, B->d1->I1, alpha);
  C->d1->I0 = hybrid_selection_average(A->d1->I0, B->d1->I0, alpha);
  printf("...Done\n");

  printf("\nFusing Gradient 0...");
  C->g->I5 = hybrid_selection_average(A->g->I5, B->g->I5, alpha);
  C->g->I4 = NULL;
  C->g->I3 = NULL;
  C->g->I2 = NULL;
  C->g->I1 = NULL;
  C->g->I0 = NULL;
  printf("...Done\n");

  printf("\nConstructing Composite Pyramid...");
  comp = construct_gradient(C);
  free_Pyramid_Gradient(C);
  printf("...Done\n");

  return(comp);
}

/* This function fuses two images by constructing gradient pyramids and
   then fusing the pyramids using hybrid fusion.  */
image * fuse_gradient_hybrid(image *i1, image *i2, double alpha)
{
  image *c;
  Pyramid_Gradient *A, *B;
  A = (Pyramid_Gradient *)malloc(sizeof(Pyramid_Gradient));
  B = (Pyramid_Gradient *)malloc(sizeof(Pyramid_Gradient));

  printf("\nCreating Pyramid A...");
  A = gen_pyramid_gradient(i1);
  printf("...Done\n");
  printf("\nCreating Pyramid B...");
  B = gen_pyramid_gradient(i2);
  printf("...Done\n");

  c = fuse_gradient_hybrid_pyramid(A, B, alpha);

  free_Pyramid_Gradient(A);
  free_Pyramid_Gradient(B);

  return(c);
}

/*****************************************************************************/
/*                    Fuse RoLp Pyramid with selection fusion                */
/*****************************************************************************/
/* This function fuses two RoLP pyramids using selection fusion  */
image * fuse_RoLP_selection_pyramid(Pyramid *A, Pyramid *B)
{
  Pyramid *C;
  image *comp;
  C = (Pyramid *)malloc(sizeof(Pyramid));

  C->I5 = selection(A->I5, B->I5);
  C->I4 = selection(A->I4, B->I4);
  C->I3 = selection(A->I3, B->I3);
  C->I2 = selection(A->I2, B->I2);
  C->I1 = selection(A->I1, B->I1);
  C->I0 = selection(A->I0, B->I0);

  comp = construct_RoLP(C);
  free_pyramid(C);

  return(comp);
}

/* This function fuses two images by constructing RoLP pyramids and then
   fusing the pyramids using selection fusion.  */
image * fuse_RoLP_selection(image *i1, image *i2)
{
  image *c;
  Pyramid *A, *B;
  A = (Pyramid *)malloc(sizeof(Pyramid));
  B = (Pyramid *)malloc(sizeof(Pyramid));

  A = gen_pyramid_RoLP(i1);
  B = gen_pyramid_RoLP(i2);

  c = fuse_RoLP_selection_pyramid(A, B);

  free_pyramid(A);
  free_pyramid(B);

  return(c);
}

/******************************************************************************/
/*                   Fuse Morph Pyramid with selection fusion                 */
/******************************************************************************/
/* This function fuses two morphological pyramids using selection fusion  */
image * fuse_morph_selection_pyramid(Pyramid *A, Pyramid *B)
{
  Pyramid *C;
  image *comp;

  C = (Pyramid *)malloc(sizeof(Pyramid));

  C->I5 = selection(A->I5, B->I5);
  C->I4 = selection(A->I4, B->I4);
  C->I3 = selection(A->I3, B->I3);
  C->I2 = selection(A->I2, B->I2);
  C->I1 = selection(A->I1, B->I1);
  C->I0 = selection(A->I0, B->I0);

  comp = construct_morph(C);
  free_pyramid(C);

  return(comp);
}

/* This function fuses two images by constructing morphological pyramids and
   then fusing the pyramids using selection fusion.  */
image * fuse_morph_selection(image *i1, image *i2)
{
  image *c;
  Pyramid *A, *B;
  A = (Pyramid *)malloc(sizeof(Pyramid));
  B = (Pyramid *)malloc(sizeof(Pyramid));

  init_sK();
  A = gen_pyramid_morph(i1);
  B = gen_pyramid_morph(i2);

  c = fuse_morph_selection_pyramid(A, B);

  free_pyramid(A);
  free_pyramid(B);

  return(c);
}

/******************************************************************************/
/*                    Fuse Morph Pyramid with hybrid fusion                   */
/******************************************************************************/
/* This function fuses two morphological pyramids using hybrid fusion  */
image * fuse_morph_hybrid_pyramid(Pyramid *A, Pyramid *B, double alpha)
{
  Pyramid *C;
  image *comp;

  C = (Pyramid *)malloc(sizeof(Pyramid));

  C->I5 = hybrid_selection_average(A->I5, B->I5, alpha);
  C->I4 = hybrid_selection_average(A->I4, B->I4, alpha);
  C->I3 = hybrid_selection_average(A->I3, B->I3, alpha);
  C->I2 = hybrid_selection_average(A->I2, B->I2, alpha);
  C->I1 = hybrid_selection_average(A->I1, B->I1, alpha);
  C->I0 = hybrid_selection_average(A->I0, B->I0, alpha);

  comp = construct_morph(C);
  free_pyramid(C);

  return(comp);
}

/* This function fuses two images by constructing morphological pyramids and
   then fusing the pyramids using hybrid fusion.  */
image * fuse_morph_hybrid(image *i1, image *i2, double alpha)
{
  image *c;
  Pyramid *A, *B;
  A = (Pyramid *)malloc(sizeof(Pyramid));
  B = (Pyramid *)malloc(sizeof(Pyramid));

  init_sK();
  A = gen_pyramid_morph(i1);
  B = gen_pyramid_morph(i2);

  c = fuse_morph_hybrid_pyramid(A, B, alpha);

  free_pyramid(A);
  free_pyramid(B);

  return(c);
}

/******************************************************************************/
/*                    Fuse RoLp Pyramid with hybrid fusion                    */
/******************************************************************************/
/* This function fuses two RoLP pyramids using hybrid fusion  */
image * fuse_RoLP_hybrid_pyramid(Pyramid *A, Pyramid *B, double alpha)
{
  Pyramid *C;
  image *comp;

  C = (Pyramid *)malloc(sizeof(Pyramid));

  C->I5 = hybrid_selection_average(A->I5, B->I5, alpha);
  C->I4 = hybrid_selection_average(A->I4, B->I4, alpha);
  C->I3 = hybrid_selection_average(A->I3, B->I3, alpha);
  C->I2 = hybrid_selection_average(A->I2, B->I2, alpha);
  C->I1 = hybrid_selection_average(A->I1, B->I1, alpha);
  C->I0 = hybrid_selection_average(A->I0, B->I0, alpha);

  comp = construct_RoLP(C);
  free_pyramid(C);

  return(comp);
}

/* This function fuses two images by constructing RoLP pyramids and
   then fusing the pyramids using hybrid fusion.  */
image * fuse_RoLP_hybrid(image *i1, image *i2, double alpha)
{
  image *c;
  Pyramid *A, *B;
  A = (Pyramid *)malloc(sizeof(Pyramid));
  B = (Pyramid *)malloc(sizeof(Pyramid));

  A = gen_pyramid_RoLP(i1);
  B = gen_pyramid_RoLP(i2);

  c = fuse_RoLP_hybrid_pyramid(A, B, alpha);

  free_pyramid(A);
  free_pyramid(B);

  return(c);
}

/****************************************************************************/
/*                  Fuse Gradient Pyramid with hybrid fusion               */
/****************************************************************************/
/* This function fuses two gradient pyramids using selection fusion  */
image * fuse_gradient_selection_pyramid(Pyramid_Gradient *A, Pyramid_Gradient *B)
{
  Pyramid_Gradient *C;
  image *comp;

  C = (Pyramid_Gradient *)malloc(sizeof(Pyramid_Gradient));
  C->d4 = (Pyramid *)malloc(sizeof(Pyramid));
  C->d3 = (Pyramid *)malloc(sizeof(Pyramid));
  C->d2 = (Pyramid *)malloc(sizeof(Pyramid));
  C->d1 = (Pyramid *)malloc(sizeof(Pyramid));
  C->g = (Pyramid *)malloc(sizeof(Pyramid));

  C->d4->I5 = NULL;
  C->d4->I4 = selection(A->d4->I4, B->d4->I4);
  C->d4->I3 = selection(A->d4->I3, B->d4->I3);
  C->d4->I2 = selection(A->d4->I2, B->d4->I2);
  C->d4->I1 = selection(A->d4->I1, B->d4->I1);
  C->d4->I0 = selection(A->d4->I0, B->d4->I0);

  C->d3->I5 = NULL;
  C->d3->I4 = selection(A->d3->I4, B->d3->I4);
  C->d3->I3 = selection(A->d3->I3, B->d3->I3);
  C->d3->I2 = selection(A->d3->I2, B->d3->I2);
  C->d3->I1 = selection(A->d3->I1, B->d3->I1);
  C->d3->I0 = selection(A->d3->I0, B->d3->I0);

  C->d2->I5 = NULL;
  C->d2->I4 = selection(A->d2->I4, B->d2->I4);
  C->d2->I3 = selection(A->d2->I3, B->d2->I3);
  C->d2->I2 = selection(A->d2->I2, B->d2->I2);
  C->d2->I1 = selection(A->d2->I1, B->d2->I1);
  C->d2->I0 = selection(A->d2->I0, B->d2->I0);

  C->d1->I5 = NULL;
  C->d1->I4 = selection(A->d1->I4, B->d1->I4);
  C->d1->I3 = selection(A->d1->I3, B->d1->I3);
  C->d1->I2 = selection(A->d1->I2, B->d1->I2);
  C->d1->I1 = selection(A->d1->I1, B->d1->I1);
  C->d1->I0 = selection(A->d1->I0, B->d1->I0);

  C->g->I5 = selection(A->g->I5, B->g->I5);
  C->g->I4 = NULL;
  C->g->I3 = NULL;
  C->g->I2 = NULL;
  C->g->I1 = NULL;
  C->g->I0 = NULL;

  comp = construct_gradient(C);
  free_Pyramid_Gradient(C);

  return(comp);
}

/* This function fuses two images by constructing gradient pyramids and
   then fusing the pyramids using selection fusion.  */
image * fuse_gradient_selection(image *i1, image *i2)
{
  image *c;
  Pyramid_Gradient *A, *B;
  A = (Pyramid_Gradient *)malloc(sizeof(Pyramid_Gradient));
  B = (Pyramid_Gradient *)malloc(sizeof(Pyramid_Gradient));

  printf("\nCreating Pyramid A...");
  A = gen_pyramid_gradient(i1);
  printf("...Done\n");
  printf("\nCreating Pyramid B...");
  B = gen_pyramid_gradient(i2);
  printf("...Done\n");

  c = fuse_gradient_selection_pyramid(A, B);

  free_Pyramid_Gradient(A);
  free_Pyramid_Gradient(B);

  return(c);
}

#endif
```

```
/************************************* image_p.cpp ******************************/
/*   This file contains the code used for image processing routines such as    */
/*   image addition, subtraction, multiplication, division, etc.                */
/********************************************************************************/
#ifndef IMAGE
#define IMAGE

#include "thesis.h"
#include "files.cpp"

/* Type Definitions  */
typedef struct Pyramid {
  image *I0, *I1, *I2, *I3, *I4, *I5;
} Pyramid;

/* Image Processing */
image * reduce(image *g);
image * expand(image *g);
image * subtract(image *i1, image *i2);
image * add(image *i1, image *i2);
image * max_comp(image *i1, image *i2);
image * divide(image *i1, image *i2);
image * multiply(image *i1, image *i2);
void free_pyramid(Pyramid *p);

/********************************************************************************/
/*                       IMAGE PROCESSING STUFF                                 */
/********************************************************************************/
/*  The Gaussian kernel, w.  */
double w(int m, int n)
{
  double wm, wn, answer;
  if(m == 0)             wm = A_REDUCE;
  else if (abs(m) == 1)  wm = 0.25;
  else                   wm = 0.25 - A_REDUCE/2;

  if(n == 0)             wn = A_REDUCE;
  else if (abs(n) == 1)  wn = 0.25;
  else                   wn = 0.25 - A_REDUCE/2;

  answer = wm*wn;
  return(answer);
}

/* Returns a single sample located at (i, j) for the REDUCED image of pic
   color is the indicator for red, green, or blue pixels.  */
int reducepixel(image * pic, int i, int j, int color)
{
  int m, n, index, tx, ty;
  double tsum = 0.0;
  for(m = -2; m < 3; m++)  {
    for(n = -2; n < 3; n++)  {
      tx = 2*i + m;
      if(tx < 0) tx = -tx;
      if(tx >= pic->xsize)   tx = 2*pic->xsize - tx - 1;

      ty = 2*j + n;
      if(ty < 0) ty = -ty;
      if(ty >= pic->ysize)   ty = 2*pic->ysize - ty - 1;

      index = ty*pic->xsize + tx;
      tsum += w(m,n) * pic->pixel[index][color];
    }
  }
  return((int)tsum);
}

/* Filters and subsamples the image g to get the next level in the
   Gaussian pyramid.  */
```

```
image * reduce(image *g)
{
  image *newpic;
  int xnew, ynew, x, y, index;
  newpic = (image *)malloc(sizeof(image));
  xnew = (g->xsize - 1)/2 + 1;
  ynew = (g->ysize - 1)/2 + 1;
  newpic->pixel = (Pixel *)malloc(xnew*ynew*sizeof(Pixel));
  for(x = 0; x < xnew; x++)  {
    for(y = 0; y < ynew; y++)  {
      index = y*xnew + x;
      newpic->pixel[index][0] = reducepixel(g, x, y, 0);
      newpic->pixel[index][1] = reducepixel(g, x, y, 1);
      newpic->pixel[index][2] = reducepixel(g, x, y, 2);
    }
  }
  newpic->xsize = xnew;
  newpic->ysize = ynew;
  return(newpic);
}


/* Returns a single sample located at (i, j) for the EXPANDED image of pic
   color is the indicator for red, green, or blue pixels.  */
double expandpixel(image * pic, int i, int j, int color)
{
  int m, n, index, tx, ty;
  double tsum = 0.0;
  for(m = -2; m < 3; m++)  {
    for(n = -2; n < 3; n++)  {
      if( ((i-m)%2 == 0) && ((j-n)%2 == 0) )  {
        tx = (i-m)/2;
        if(tx < 0) tx = -tx;
        if(tx >= pic->xsize)   tx = 2*pic->xsize - tx - 1;

        ty = (j - n)/2;
        if(ty < 0) ty = -ty;
        if(ty >= pic->ysize)   ty = 2*pic->ysize - ty - 1;

        index = ty*pic->xsize + tx;
        tsum += w(m,n) * pic->pixel[index][color];
      }
    }
  }
  return(4*tsum);
}

/* Upsamples and interpolates the image g to get a lower resolution image
   of the previoius level in the pyramid*/
image * expand(image *g)
{
  image *newpic;
  int xnew, ynew, x, y, index;
  newpic = (image *)malloc(sizeof(image));
  xnew = (g->xsize - 1)*2 + 1;
  ynew = (g->ysize - 1)*2 + 1;
  newpic->pixel = (Pixel *)malloc(xnew*ynew*sizeof(Pixel));
  for(x = 0; x < xnew; x++)  {
    for(y = 0; y < ynew; y++)  {
      index = y*xnew + x;
      newpic->pixel[index][0] = expandpixel(g, x, y, 0);
      newpic->pixel[index][1] = expandpixel(g, x, y, 1);
      newpic->pixel[index][2] = expandpixel(g, x, y, 2);
    }
  }
  newpic->xsize = xnew;
  newpic->ysize = ynew;
  return(newpic);
}
```

```
/*  This function returns the absolut value of the difference between
    image i1 and image i2.  */
image * subtract(image *i1, image *i2)
{
  image *newpic;
  int xnew, ynew, x, y, index;
  if(i1->xsize == i2->xsize && i1->ysize == i2->ysize)  {
    xnew = i1->xsize;
    ynew = i2->ysize;
    newpic = (image *)malloc(sizeof(image));
    newpic->pixel = (Pixel *)malloc(xnew*ynew*sizeof(Pixel));
    for(x = 0; x < xnew; x++)  {
      for(y = 0; y < ynew; y++)  {
        index = y*xnew + x;
        newpic->pixel[index][0] = abs(i1->pixel[index][0] - i2->pixel[index][0]);
        newpic->pixel[index][1] = abs(i1->pixel[index][1] - i2->pixel[index][1]);
        newpic->pixel[index][2] = abs(i1->pixel[index][2] - i2->pixel[index][2]);
      }
    }
    newpic->xsize = xnew;
    newpic->ysize = ynew;
    return(newpic);
  }
  else return(NULL);
}

/*  This function returns the difference between image i1 and image i2.  */
image * subtract_sign(image *i1, image *i2)
{
  image *newpic;
  int xnew, ynew, x, y, index;
  if(i1->xsize == i2->xsize && i1->ysize == i2->ysize)  {
    xnew = i1->xsize;
    ynew = i2->ysize;
    newpic = (image *)malloc(sizeof(image));
    newpic->pixel = (Pixel *)malloc(xnew*ynew*sizeof(Pixel));
    for(x = 0; x < xnew; x++)  {
      for(y = 0; y < ynew; y++)  {
        index = y*xnew + x;
        newpic->pixel[index][0] = i1->pixel[index][0] - i2->pixel[index][0];
        newpic->pixel[index][1] = i1->pixel[index][1] - i2->pixel[index][1];
        newpic->pixel[index][2] = i1->pixel[index][2] - i2->pixel[index][2];
      }
    }
    newpic->xsize = xnew;
    newpic->ysize = ynew;
    return(newpic);
  }
  else return(NULL);
}

/*  This function returns the sum of image i1 and image i2.  */
image * add_sign(image *i1, image *i2)
{
  image *newpic;
  int xnew, ynew, x, y, index;
  if(i1->xsize == i2->xsize && i1->ysize == i2->ysize)  {
    xnew = i1->xsize;
    ynew = i2->ysize;
    newpic = (image *)malloc(sizeof(image));
    newpic->pixel = (Pixel *)malloc(xnew*ynew*sizeof(Pixel));
    for(y = 0; y < ynew; y++)  {
      for(x = 0; x < xnew; x++)  {
        index = y*xnew + x;
        newpic->pixel[index][0] = i1->pixel[index][0] + i2->pixel[index][0];
        newpic->pixel[index][1] = i1->pixel[index][1] + i2->pixel[index][1];
        newpic->pixel[index][2] = i1->pixel[index][2] + i2->pixel[index][2];
      }
    }
    newpic->xsize = xnew;
    newpic->ysize = ynew;
    return(newpic);
  }
  else return(NULL);
}

/*  This function frees the memory used by a pyramid.  */
void free_pyramid(Pyramid *p)
{
  if(p != NULL)  {
    free_image(p->I0);
    free_image(p->I1);
    free_image(p->I2);
    free_image(p->I3);
    free_image(p->I4);
    free_image(p->I5);
    free(p);
    p = NULL;
  }
  return;
}


/*  This function returns the ratio of image i1 and image i2.  */
image * divide(image *i1, image *i2)
{
  image *newpic;
  int xnew, ynew, x, y, index;
  if(i1->xsize == i2->xsize && i1->ysize == i2->ysize)  {
    xnew = i1->xsize;
    ynew = i2->ysize;
    newpic = (image *)malloc(sizeof(image));
    newpic->pixel = (Pixel *)malloc(xnew*ynew*sizeof(Pixel));
    for(x = 0; x < xnew; x++)  {
      for(y = 0; y < ynew; y++)  {
        index = y*xnew + x;
        if(i2->pixel[index][0] != 0)  {
          newpic->pixel[index][0] = (i1->pixel[index][0] / i2->pixel[index][0]);
        }
        else  {
          newpic->pixel[index][0] = 1;
        }
        if(i2->pixel[index][1] != 0)  {
          newpic->pixel[index][1] = (i1->pixel[index][1] / i2->pixel[index][1]);
        }
        else  {
          newpic->pixel[index][1] = 1;
        }
        if(i2->pixel[index][2] != 0)  {
          newpic->pixel[index][2] = (i1->pixel[index][2] / i2->pixel[index][2]);
        }
        else  {
          newpic->pixel[index][2] = 1;
        }
      }
    }
    newpic->xsize = xnew;
    newpic->ysize = ynew;
    return(newpic);
  }
  else return(NULL);
}


/*  This function returns the product of image i1 and image i2.  */
image * multiply(image *i1, image *i2)
{
  image *newpic;
  int xnew, ynew, x, y, index;
  if(i1->xsize == i2->xsize && i1->ysize == i2->ysize)  {
    xnew = i1->xsize;
    ynew = i2->ysize;
    newpic = (image *)malloc(sizeof(image));
    newpic->pixel = (Pixel *)malloc(xnew*ynew*sizeof(Pixel));
    for(x = 0; x < xnew; x++)  {
      for(y = 0; y < ynew; y++)  {
        index = y*xnew + x;
        newpic->pixel[index][0] = (i1->pixel[index][0] * i2->pixel[index][0]);
        newpic->pixel[index][1] = (i1->pixel[index][1] * i2->pixel[index][1]);
        newpic->pixel[index][2] = (i1->pixel[index][2] * i2->pixel[index][2]);
      }
    }
    newpic->xsize = xnew;
    newpic->ysize = ynew;
    return(newpic);
  }
  else return(NULL);
}

/* This function creates an image structure from the bands of other images.
   i1, i2, and i3 are the images used, and c1, c2, and c3 are the bands that
   will form the new red, green, and blue bands.  */
image * spec_bands(image *i1, int c1, image *i2, int c2, image *i3, int c3)
{
  image *newpic;
  int xnew, ynew, x, y, index;
  if(i1->xsize == i2->xsize && i1->ysize == i2->ysize &&
     i2->xsize == i3->xsize && i2->ysize == i3->ysize)  {
    xnew = i1->xsize;
    ynew = i2->ysize;
    newpic = (image *)malloc(sizeof(image));
    newpic->pixel = (Pixel *)malloc(xnew*ynew*sizeof(Pixel));
    for(x = 0; x < xnew; x++)  {
      for(y = 0; y < ynew; y++)  {
        index = y*xnew + x;
        newpic->pixel[index][0] = i1->pixel[index][c1];
        newpic->pixel[index][1] = i2->pixel[index][c2];
        newpic->pixel[index][2] = i3->pixel[index][c3];
      }
    }
    newpic->xsize = xnew;
    newpic->ysize = ynew;
    return(newpic);
  }
  else return(NULL);
}

/* This function creates a ppm file from the bands of other ppm files.
   The new image contains the red band of n1, the green band of n2, and
   the blue band of n3.  */
void mergebands(char * out, char * n1, char * n2, char * n3)
{
  int xsize, ysize, x, y;
  FILE * infile1, *infile2, *infile3, *outfile;
  int red, green, blue, t1, t2;
  initfile(&infile1, n1, &xsize, &ysize);
  initfile(&infile2, n2, &xsize, &ysize);
  initfile(&infile3, n3, &xsize, &ysize);
  outfile = fopen(out, "w");
  fprintf(outfile, "P3\n# Created by Ted Meek\n");
  fprintf(outfile, "%d %d %d\n", xsize, ysize, 255);
  for(y = 0; y < ysize; y++)  {
    for(x = 0; x < xsize; x++)  {
      fscanf(infile1, "%d%d%d", &red, &t1, &t2);
      fscanf(infile2, "%d%d%d", &t1, &green, &t2);
      fscanf(infile3, "%d%d%d", &t1, &t2, &blue);
      fprintf(outfile, "%d %d %d ", red, green, blue);
    }
  }
  fclose(infile1);
  fclose(infile2);
  fclose(infile3);
  fclose(outfile);
}

/*  This function returns the product of image i1 and the constand c.  */
image * multiply_const(image *i1, double c)
{
  image *newpic;
  int xnew, ynew, x, y, index;
    xnew = i1->xsize;
    ynew = i1->ysize;
    newpic = (image *)malloc(sizeof(image));
    newpic->pixel = (Pixel *)malloc(xnew*ynew*sizeof(Pixel));
    for(x = 0; x < xnew; x++)  {
      for(y = 0; y < ynew; y++)  {
        index = y*xnew + x;
        newpic->pixel[index][0] = (i1->pixel[index][0] * c);
        newpic->pixel[index][1] = (i1->pixel[index][1] * c);
        newpic->pixel[index][2] = (i1->pixel[index][2] * c);
        for(int i = 0; i < 3; i++)  {
          if(newpic->pixel[index][i] > 255) newpic->pixel[index][i] = 255;
        }
      }
    }
    newpic->xsize = xnew;
    newpic->ysize = ynew;
    return(newpic);
}

#endif
```

```
/********************************* gaussian_n.cpp ****************************/
/*   This file contains the code for generating a Gaussian pyramid for a given */
/*   image.  The pyramid depth is six levels.                                  */
/******************************************************************************/
#ifndef GAUSSIAN
#define GAUSSIAN

#include "thesis.h"
#include "files.cpp"
#include "image_p.cpp"

/*********************************************************************************/
/*                              Gaussian                                         */
/*********************************************************************************/
/* This functin generates a six level Gaussian pyramid for the given image */
Pyramid * gen_pyramid_gaussian(image *g)
{
  Pyramid *p;

  p = (Pyramid *)malloc(sizeof(Pyramid));

  p->I0 = resize(g, 5);
  p->I1 = reduce(p->I0);
  p->I2 = reduce(p->I1);
  p->I3 = reduce(p->I2);
  p->I4 = reduce(p->I3);
  p->I5 = reduce(p->I4);

  return(p);
}

#endif
```

```
/********************************** leplace_n.cpp *****************************/
/*   This file contains the code for generating a Leplacian pyramid for a     */
/*   given image.  The pyramid depth is six levels.                           */
/******************************************************************************/
#ifndef LEPLACE
#define LEPLACE

#include "thesis.h"
#include "gaussian_n.cpp"

/* Type Definitions  */

/* Image Processing */
Pyramid * gen_pyramid_leplace(image *g);
image * construct_leplace(Pyramid *p);

/*********************************************************************************/
/*                              Leplace                                          */
/*********************************************************************************/
Pyramid * gen_pyramid_leplace(image *i)
{
  Pyramid *g, *p;
  image * temp;

  p = (Pyramid *)malloc(sizeof(Pyramid));
  g = gen_pyramid_gaussian(i);

  p->I5 = copy_image(g->I5);
  temp = expand(g->I5);
  p->I4 = subtract_sign(g->I4, temp);
  free_image(temp);
  temp = expand(g->I4);
  p->I3 = subtract_sign(g->I3, temp);
  free_image(temp);
  temp = expand(g->I3);
  p->I2 = subtract_sign(g->I2, temp);
  free_image(temp);
  temp = expand(g->I2);
  p->I1 = subtract_sign(g->I1, temp);
  free_image(temp);
  temp = expand(g->I1);
  p->I0 = subtract_sign(g->I0, temp);
  free_image(temp);

  return(p);
}


image * construct_leplace(Pyramid *L)
{
  image *temp, *i;
  Pyramid *p;

  p = (Pyramid *)malloc(sizeof(Pyramid));

  temp = expand(L->I5);  p->I4 = add_sign(temp, L->I4);  free(temp);
  temp = expand(p->I4);  p->I3 = add_sign(temp, L->I3);  free(temp);
  temp = expand(p->I3);  p->I2 = add_sign(temp, L->I2);  free(temp);
  temp = expand(p->I2);  p->I1 = add_sign(temp, L->I1);  free(temp);
  temp = expand(p->I1);  p->I0 = add_sign(temp, L->I0);  free(temp);
i = copy_image(p->I0);

  return(i);
}

#endif
```

```
/********************************* RoLP_n.cpp *********************************/
/*   This file contains the code for generating a RoLP pyramid for a given     */
/*   image.  The pyramid depth is six levels.                                  */
/******************************************************************************/
#ifndef ROLP
#define ROLP
#include "gaussian_n.cpp"

Pyramid * gen_pyramid_RoLP(image *g);
image * construct_RoLP(Pyramid *p);
image * max_RoLP(image *i1, image *i2);

/********************************************************************************/
/*                                     RoLP                                     */
/********************************************************************************/
/* This functin generates a six level RoLP pyramid for the given image */
Pyramid * gen_pyramid_RoLP(image *i)
{
  Pyramid *p, *g;
  image * temp;

  p = (Pyramid *)malloc(sizeof(Pyramid));
  g = gen_pyramid_gaussian(i);

  p->I5 = copy_image(g->I5);
  temp = expand(g->I5);  p->I4 = divide(g->I4, temp);  free_image(temp);
  temp = expand(g->I4);  p->I3 = divide(g->I3, temp);  free_image(temp);
  temp = expand(g->I3);  p->I2 = divide(g->I2, temp);  free_image(temp);
  temp = expand(g->I2);  p->I1 = divide(g->I1, temp);  free_image(temp);
  temp = expand(g->I1);  p->I0 = divide(g->I0, temp);  free_image(temp);
  return(p);
}

/*  This function returns the source image from a RoLP pyramid.  */
image * construct_RoLP(Pyramid *r)
{
  image *temp, *i;
  Pyramid *p;

  p = (Pyramid *)malloc(sizeof(Pyramid));
  i = (image *)malloc(sizeof(image));

  if(r->I5 != NULL)  {
    p->I5 = copy_image(r->I5);
    temp = expand(p->I5);
    p->I4 = multiply(r->I4, temp);
    free(temp);
  }
  else  {
    p->I4 = copy_image(r->I4);
  }

  temp = expand(p->I4);
  p->I3 = multiply(r->I3, temp);
  free(temp);
  temp = expand(p->I3);
  p->I2 = multiply(r->I2, temp);
  free(temp);
  temp = expand(p->I2);
  p->I1 = multiply(r->I1, temp);
  free(temp);
  temp = expand(p->I1);
  p->I0 = multiply(r->I0, temp);
  i = multiply(r->I0, temp);
  free(temp);
  return(i);
}

#endif
```

```
/*********************************** gradient_n.cpp ***************************/
/*   This file contains the code for generating a gradient pyramid for a given */
/*   image.  The pyramid depth is six levels.  The gradient pyramid is         */
/*   actually four pyramids so a new structure is also defined and used.       */
/******************************************************************************/
#ifndef GRADIENT
#define GRADIENT

#include "thesis.h"
#include "convolve_n.cpp"
#include "gaussian_n.cpp"

/* Type Definitions  */
typedef struct Pyramid_Gradient {
  Pyramid *g, *d1, *d2, *d3, *d4;
} Pyramid_Gradient;

/* Image Processing */
void free_Pyramid_Gradient(Pyramid_Gradient *p);
Pyramid_Gradient * gen_pyramid_gradient(image *g);
image * construct_gradient(Pyramid_Gradient *p);

/*****************************************************************************/
/*                                     Gradient                              */
/*****************************************************************************/
/*  This function frees the memory used for a gradient pyramid.  */
void free_Pyramid_Gradient(Pyramid_Gradient *p)
{
  if(p != NULL)  {
    free_pyramid(p->g);
    free_pyramid(p->d1);
    free_pyramid(p->d2);
    free_pyramid(p->d3);
    free_pyramid(p->d4);
    free(p);
    p = NULL;
  }
  return;
}

/*  This function generates the four seperate pyramids which for the gradient pyramid. */
void gen_gradient(image *g, image **g1, image **g2, image **g3, image **g4)
{
  image * temp, *temp2;

  temp = conv_w_g_dot_image(g);
  temp2 = add_sign(temp, g);
  free_image(temp);
  (*g1) = conv_d1_image(temp2, 1);
  (*g2) = conv_d2_image(temp2, 1);
  (*g3) = conv_d3_image(temp2, 1);
  (*g4) = conv_d4_image(temp2, 1);
  free_image(temp2);

  return;
}

/*  This function generates the gradient pyramid for a given image.  */
Pyramid_Gradient * gen_pyramid_gradient(image *i)
{
  Pyramid_Gradient *p;

  p = (Pyramid_Gradient *)malloc(sizeof(Pyramid_Gradient));
  p->d1 = (Pyramid *)malloc(sizeof(Pyramid));
  p->d2 = (Pyramid *)malloc(sizeof(Pyramid));
  p->d3 = (Pyramid *)malloc(sizeof(Pyramid));
  p->d4 = (Pyramid *)malloc(sizeof(Pyramid));
  p->g = (Pyramid *)malloc(sizeof(Pyramid));
  p->g = gen_pyramid_gaussian(i);
  gen_gradient(p->g->I0, &p->d1->I0, &p->d2->I0, &p->d3->I0, &p->d4->I0);
  gen_gradient(p->g->I1, &p->d1->I1, &p->d2->I1, &p->d3->I1, &p->d4->I1);
  gen_gradient(p->g->I2, &p->d1->I2, &p->d2->I2, &p->d3->I2, &p->d4->I2);
  gen_gradient(p->g->I3, &p->d1->I3, &p->d2->I3, &p->d3->I3, &p->d4->I3);
  gen_gradient(p->g->I4, &p->d1->I4, &p->d2->I4, &p->d3->I4, &p->d4->I4);
  gen_gradient(p->g->I5, &p->d1->I5, &p->d2->I5, &p->d3->I5, &p->d4->I5);
  return(p);
}
/*  This function returns the source image from a gradient pyramid.  */
image * construct_gradient(Pyramid_Gradient *p)
{
  image *g3, *g2, *g1, *g0, *g4;
  image *l3, *l2, *l1, *l0, *l4;
  image *lx1, *lx2, *lx3, *lx4;
  image *lf1, *lf2, *lf3, *lf0, *lf4;

  lx1 = gen_gradient_FSD_d1(p->d1->I4);
  lx2 = gen_gradient_FSD_d2(p->d2->I4);
  lx3 = gen_gradient_FSD_d3(p->d3->I4);
  lx4 = gen_gradient_FSD_d4(p->d4->I4);
  lf4 = addl1tol4(lx1, lx2, lx3, lx4);
  free_image(lx1);  free_image(lx2);  free_image(lx3);  free_image(lx4);

  lx1 = gen_gradient_FSD_d1(p->d1->I3);
  lx2 = gen_gradient_FSD_d2(p->d2->I3);
  lx3 = gen_gradient_FSD_d3(p->d3->I3);
  lx4 = gen_gradient_FSD_d4(p->d4->I3);
  lf3 = addl1tol4(lx1, lx2, lx3, lx4);
  free_image(lx1);  free_image(lx2);  free_image(lx3);  free_image(lx4);

  lx1 = gen_gradient_FSD_d1(p->d1->I2);
  lx2 = gen_gradient_FSD_d2(p->d2->I2);
  lx3 = gen_gradient_FSD_d3(p->d3->I2);
  lx4 = gen_gradient_FSD_d4(p->d4->I2);
  lf2 = addl1tol4(lx1, lx2, lx3, lx4);
  free_image(lx1);  free_image(lx2);  free_image(lx3);  free_image(lx4);

  lx1 = gen_gradient_FSD_d1(p->d1->I1);
  lx2 = gen_gradient_FSD_d2(p->d2->I1);
  lx3 = gen_gradient_FSD_d3(p->d3->I1);
  lx4 = gen_gradient_FSD_d4(p->d4->I1);
  lf1 = addl1tol4(lx1, lx2, lx3, lx4);
  free_image(lx1);  free_image(lx2);  free_image(lx3);  free_image(lx4);

  lx1 = gen_gradient_FSD_d1(p->d1->I0);
  lx2 = gen_gradient_FSD_d2(p->d2->I0);
  lx3 = gen_gradient_FSD_d3(p->d3->I0);
  lx4 = gen_gradient_FSD_d4(p->d4->I0);
  lf0 = addl1tol4(lx1, lx2, lx3, lx4);
  free_image(lx1);  free_image(lx2);  free_image(lx3);  free_image(lx4);

  l4 = FSD_to_RE(lf4);
  l3 = FSD_to_RE(lf3);
  l2 = FSD_to_RE(lf2);
  l1 = FSD_to_RE(lf1);
  l0 = FSD_to_RE(lf0);

  g4 = add_sign(l4, expand_g(p->g->I5));
  g3 = add_sign(l3, expand_g(g4));
  g2 = add_sign(l2, expand_g(g3));
  g1 = add_sign(l1, expand_g(g2));
  g0 = add_sign(l0, expand_g(g1));

  free_image(g1);  free_image(g2);  free_image(g3);  free_image(g4);
  free_image(l3);  free_image(l2);  free_image(l1);  free_image(l0);
  return(g0);
}
#endif
```

```
/********************************** morph_n.cpp *******************************/
/*   This file contains the code for generating a morphological pyramid for a  */
/*   given image.  The pyramid depth is six levels.                             */
/******************************************************************************/
#ifndef MORPH
#define MORPH

#include "thesis.h"
#include "files.cpp"
#include "image_p.cpp"

/* Type Definitions  */
typedef struct Struct_Elem {
  int xsize, ysize, xcen, ycen;
  Pixel *pixel;
} Struct_Elem;

/* Image Processing */
Pyramid * gen_pyramid_morph(image *g);
image * construct_morph(Pyramid *p);
image * expand_m(image *g);
image * reduce_m(image *g);

/* Global Variables */
Struct_Elem K;

/******************************************************************************/
/*                      STRUCTURING ELEMENT STUFF                             */
/******************************************************************************/
/*  Defines the structuring element as a 3-by-3 brick.  */
void init_K()
{
  K.xsize = 3;  K.ysize = 3;
  K.ycen = 1;  K.xcen = 1;
  K.pixel = (Pixel *)malloc(K.xsize*K.ysize*sizeof(Pixel));
  K.pixel[0][0] = 1;  K.pixel[1][0] = 1;  K.pixel[2][0] = 1;
  K.pixel[3][0] = 1;  K.pixel[4][0] = 1;  K.pixel[5][0] = 1;
  K.pixel[6][0] = 1;  K.pixel[7][0] = 1;  K.pixel[8][0] = 1;

  K.pixel[0][1] = 1;  K.pixel[1][1] = 1;  K.pixel[2][1] = 1;
  K.pixel[3][1] = 1;  K.pixel[4][1] = 1;  K.pixel[5][1] = 1;
  K.pixel[6][1] = 1;  K.pixel[7][1] = 1;  K.pixel[8][1] = 1;

  K.pixel[0][2] = 1;  K.pixel[1][2] = 1;  K.pixel[2][2] = 1;
  K.pixel[3][2] = 1;  K.pixel[4][2] = 1;  K.pixel[5][2] = 1;
  K.pixel[6][2] = 1;  K.pixel[7][2] = 1;  K.pixel[8][2] = 1;
}

/*  Defines the structuring element as a 2-by-2 brick.  */
void init_sK()
{
  K.xsize = 2;  K.ysize = 2;
  K.ycen = 1;  K.xcen = 1;
  K.pixel = (Pixel *)malloc(K.xsize*K.ysize*sizeof(Pixel));
  K.pixel[0][0] = 1;  K.pixel[1][0] = 1;
  K.pixel[2][0] = 1;  K.pixel[3][0] = 1;

  K.pixel[0][1] = 1;  K.pixel[1][1] = 1;
  K.pixel[2][1] = 1;  K.pixel[3][1] = 1;

  K.pixel[0][2] = 1;  K.pixel[1][2] = 1;
  K.pixel[2][2] = 1;  K.pixel[3][2] = 1;
}

/*  Defines the structuring element as a 10-by-10 brick.  */
void init_bigK()
{
  int index;
  K.xsize = 10;  K.ysize = 10;
  K.ycen = 5;  K.xcen = 5;
  K.pixel = (Pixel *)malloc(K.xsize*K.ysize*sizeof(Pixel));
  for(int i = 0; i < K.xsize; i++)  {
    for(int j = 0; j < K.ysize; j++)  {
      index = j*K.xsize + i;
      K.pixel[index][0] = 1;  K.pixel[index][1] = 1;  K.pixel[index][2] = 1;
    }
  }
}

/********************************************************************************/
/*                          IMAGE PROCESSING STUFF                             */
/********************************************************************************/
/*  Returns a single pixel value at the (i, j) location for an image being
    dilated.  */
double sample_dilate(image * f, Struct_Elem *k, int i, int j, int c)
{
  int x, y, kx, ky, m, n;
  double sum, max = -1.0;

  for(m = 0; m < k->xsize; m++)  {
    for(n = 0; n < k->ysize; n++)  {
      kx = m - k->xcen;
      ky = n - k->ycen;
      x = i - kx;
      y = j - ky;
      if(x < 0 || y < 0 || x >= f->xsize || y >= f->ysize) continue;
      sum = f->pixel[y*f->xsize + x][c] + k->pixel[n*k->xsize + m][c];
      if(sum > max)  max = sum;
    }
  }

  return(max);
}

/*  Dilates the image f by the structuring element K.  */
image * dilate(image *f, Struct_Elem *k)
{
  image *newpic;
  int x, y, index;
  newpic = (image *)malloc(sizeof(image));
  newpic->xsize = f->xsize;
  newpic->ysize = f->ysize;
  newpic->pixel = (Pixel *)malloc(newpic->xsize*newpic->ysize*sizeof(Pixel));
  for(y = 0; y < newpic->ysize; y++)  {
    for(x = 0; x < newpic->xsize; x++)  {
      index = y*newpic->xsize + x;
      newpic->pixel[index][0] = sample_dilate(f, k, x, y, 0);
      newpic->pixel[index][1] = sample_dilate(f, k, x, y, 1);
      newpic->pixel[index][2] = sample_dilate(f, k, x, y, 2);
    }
  }
  return(newpic);
}

/*  Returns a single pixel value at the (i, j) location for an image being
    eroded.  */
double sample_erode(image * f, Struct_Elem *k, int i, int j, int c)
{
  int x, y, m, n, kx, ky;
  double sum, min = 1000000;

  for(m = 0; m < k->xsize; m++)  {
    for(n = 0; n < k->ysize; n++)  {
      kx = m - k->xcen;
      ky = n - k->ycen;
      x = i + kx;
      y = j + ky;
      if(x < 0 || y < 0 || x >= f->xsize || y >= f->ysize) continue;
      sum = f->pixel[y*f->xsize + x][c] - k->pixel[n*k->xsize + m][c];
      if(sum < min)  min = sum;
    }
  }
  return(min);
}

/*  Erodes the image f by the structuring element K.  */
image * erode(image *f, Struct_Elem *k)
{
  image *newpic;
  int x, y, index;
  newpic = (image *)malloc(sizeof(image));
  newpic->xsize = f->xsize;
  newpic->ysize = f->ysize;
  newpic->pixel = (Pixel *)malloc(newpic->xsize*newpic->ysize*sizeof(Pixel));
  for(y = 0; y < newpic->ysize; y++)  {
    for(x = 0; x < newpic->xsize; x++)  {
      index = y*newpic->xsize + x;
      newpic->pixel[index][0] = sample_erode(f, k, x, y, 0);
      newpic->pixel[index][1] = sample_erode(f, k, x, y, 1);
      newpic->pixel[index][2] = sample_erode(f, k, x, y, 2);
    }
  }
  return(newpic);
}

/********************************************************************************/
/*                              FILTERS                                         */
/********************************************************************************/
/*  Returns the image f after the application of the opening morphological
    filter that uses the structuring element K.  */
image * open(image *f)
{
  image *temp, *temp2;
  temp = erode(f, &K);
  temp2 = dilate(temp, &K);
  free_image(temp);
  return(temp2);
}

/*  Returns the image f after the application of the closing morphological
    filter that uses the structuring element K.  */
image * close(image *f)
{
  image *temp, *temp2;
  temp = dilate(f, &K);     //  savefile(temp, "g3rod.ppm");
  temp2 = erode(temp, &K); //  savefile(temp2, "g3rode.ppm");
  free_image(temp);
  return(temp2);
}

/*  Returns the image f after the application of both the closing and opening
    morphological filters.  */
image * FCO(image *f)
{
  image *temp, *temp2;
  temp = close(f);     //  savefile(temp, "g3ec_b.ppm");
  temp2 = open(temp); //  savefile(temp2, "g3eo_b.ppm");
  free_image(temp);
  return(temp2);
}

/*  Returns the image f after the application of both the opening and closing
    morphological filters.  */
image * FOC(image *f)
{
  image *temp, *temp2;
  temp = open(f);       //  savefile(temp, "g3o_b.ppm");
  temp2 = close(temp);  //  savefile(temp2, "g3oc_b.ppm");
  free_image(temp);
  return(temp2);
}

/*  Returns the image f after an expand operation and closing filter.  */
image * C_expand(image *f)
{
  image *temp, *temp2;
  temp = expand_m(f);
  temp2 = close(temp);
  free_image(temp);
  return(temp2);
}

/*  Returns the image f after an expand operation and dilation.  */
image * D_expand(image *f)
{
  image *temp, *temp2;
  temp = expand_m(f);
  temp2 = dilate(temp, &K);
  free_image(temp);
  return(temp2);
}

/*  Returns the image f after an expand operation and a closeing
    then opening filter.  */
image * FCO_expand(image *f)
{
  image *temp, *temp2;
  temp = expand_m(f);
  temp2 = FCO(temp);
  free_image(temp);
  return(temp2);
}

/*  Returns the image f after an expand operation and an opening
    then closeing filter.  */
image * FOC_expand(image *f)
{
  image *temp, *temp2;
  temp = FOC(f);            //  savefile(temp, "g3foc.ppm");
  temp2 = expand_m(temp); //  savefile(temp2, "g3foce.ppm");
  free_image(temp);
  return(temp2);
}

/*  Generates the next higher level of a morphological pyramid.  */
image *PC(image *f)
{
  image *temp, *temp2;
  temp = reduce_m(f);
  temp2 = FCO(temp);
  free_image(temp);
  return(temp2);
}

/*  Used for pyramid reconstruction to generate the level below the
    current level.  */
image *PRC(image *f, image *g)
{
  image *temp, *temp2;
  temp = C_expand(f);
  temp2 = subtract_sign(g, temp);
  free_image(temp);
  return(temp2);
}

/*******************************************************************************/
```

```
/*                              RESIZING ROUTINES                                */
/********************************************************************************/
/* Subsamples the image.  */
image * reduce_m(image *g)
{
  image *newpic;
  int xnew, ynew, x, y, index, i2;
  newpic = (image *)malloc(sizeof(image));
  xnew = (g->xsize - 1)/2 + 1;
  ynew = (g->ysize - 1)/2 + 1;
  newpic->pixel = (Pixel *)malloc(xnew*ynew*sizeof(Pixel));
  for(x = 0; x < xnew; x++)  {
    for(y = 0; y < ynew; y++)  {
      index = y*xnew + x;
      i2 = y*2*g->xsize + x*2;
      newpic->pixel[index][0] = g->pixel[i2][0];
      newpic->pixel[index][1] = g->pixel[i2][1];
      newpic->pixel[index][2] = g->pixel[i2][2];
    }
  }
  newpic->xsize = xnew;
  newpic->ysize = ynew;
  return(newpic);
}

/* Upsamples the image.  */
image * expand_m(image *g)
{
  image *newpic;
  int xnew, ynew, x, y, index, i2;
  newpic = (image *)malloc(sizeof(image));
  xnew = (g->xsize - 1)*2 + 1;
  ynew = (g->ysize - 1)*2 + 1;
  newpic->pixel = (Pixel *)malloc(xnew*ynew*sizeof(Pixel));
  for(x = 0; x < xnew; x++)  {
    for(y = 0; y < ynew; y++)  {
      index = y*xnew + x;
      if(x%2 == 0 && y%2 == 0)  {
        i2 = y*g->xsize/2 + x/2;
        newpic->pixel[index][0] = g->pixel[i2][0];
        newpic->pixel[index][1] = g->pixel[i2][1];
        newpic->pixel[index][2] = g->pixel[i2][2];
      }
      else  {
        newpic->pixel[index][0] = 0;
        newpic->pixel[index][1] = 0;
        newpic->pixel[index][2] = 0;
      }
    }
  }
  newpic->xsize = xnew;
  newpic->ysize = ynew;
  return(newpic);
}

/*********************************************************************************/
/*                                     Morph                                     */
/*********************************************************************************/
/* This functin generates a six level morphological pyramid for the given image */
Pyramid * gen_pyramid_morph(image *g)
{
  Pyramid *p;
  image * temp, *i0, *i1, *i2, *i3, *i4, *i5;

  p = (Pyramid *)malloc(sizeof(Pyramid));

  // A regular resize instead of resize_nn may be appropriate.
  // resize_nn keeps the edges of the box from bluring.
  temp = resize_nn(g, 5);  i0 = close(temp);  free_image(temp);
  i1 = PC(i0);  //  savefile(i1, "g1.ppm");
  i2 = PC(i1);  //  savefile(i2, "g2.ppm");
  i3 = PC(i2);  //  savefile(i3, "g3.ppm");
  i4 = PC(i3);  //  savefile(i4, "g4.ppm");
  i5 = PC(i4);  //  savefile(i5, "g5.ppm");

  p->I5 = i5;
  p->I4 = PRC(i5, i4);  //  savefile(p->I4, "l4.ppm");
  p->I3 = PRC(i4, i3);  //  savefile(p->I3, "l3.ppm");
  p->I2 = PRC(i3, i2);  //  savefile(p->I2, "pi2.ppm");
  p->I1 = PRC(i2, i1);  //  savefile(p->I1, "l1.ppm");
  p->I0 = PRC(i1, i0);  //  savefile(p->I0, "l0.ppm");
  free_image(i4);  free_image(i3);  free_image(i2);
  free_image(i1);  free_image(i0);

  return(p);
}

/*  This function returns the source image from a morphological pyramid.  */
image * construct_morph(Pyramid *p)
{
  image *g4e, *g3e, *g2e, *g1e, *g5e;
  image *i0, *i1, *i2, *i3, *i4;

  g5e = C_expand(p->I5);      //  savefile(g4e, "g4ec.ppm");
  i4 = add_sign(g5e, p->I4); //  savefile(i3, "i3c.ppm");
  g4e = C_expand(i4);     //  savefile(g4e, "g4ec.ppm");
  i3 = add_sign(g4e, p->I3); //  savefile(i3, "i3c.ppm");
  g3e = C_expand(i3);        //  savefile(g3e, "g3ec.ppm");
  i2 = add_sign(g3e, p->I2); //  savefile(i2, "i2n.ppm");
  g2e = C_expand(i2);        //  savefile(g2e, "g2en.ppm");
  i1 = add_sign(g2e, p->I1); //  savefile(i1, "i1n.ppm");
  g1e = C_expand(i1);        //  savefile(g1e, "g1en.ppm");
  i0 = add_sign(g1e, p->I0); //  savefile(i0, "i0n.ppm");
  free_image(g1e);  free_image(g2e);  free_image(g3e);  free_image(g4e);
  free_image(i3);   free_image(i2);   free_image(i1);

  return(i0);
}

#endif
```

```
/********************************** convolve.cpp *******************************/
/*   This file contains the code for implementing convolution for pyramid      */
/*   filtering.  It also contains code necessary to support the gradient       */
/*   pyramid                                                                   */
/*******************************************************************************/
#ifndef CONVOLVE
#define CONVOLVE

#include "thesis.h"
#include "files.cpp"
#include "image_p.cpp"

#define sqrt2 1.41421356237
#define sqrt2inv .707106781188
#define eighth .125

/* Image Processing */
image * reduce_g(image *g);
image * expand_g(image *g);


/*******************************************************************************/
/*                          GAUSSIAN KERNEL FUNCTIONS                          */
/*******************************************************************************/
/* Gaussian kernal */
double w_g(int m, int n)
{
  double wm, wn, answer;
  if(m == 0)             wm = 6;
  else if (abs(m) == 1)  wm = 4;
  else                   wm = 1;

  if(n == 0)             wn = 6;
  else if (abs(n) == 1)  wn = 4;
  else                   wn = 1;

  answer = wm*wn/256;
  return(answer);
}

/* Seperated function of Gaussian kernal */
double w_g_dot(int m, int n)
{
  double wm, wn, answer;
  if(m == 0)             wm = 2;
  else                   wm = 1;

  if(n == 0)             wn = 2;
  else                   wn = 1;

  answer = wm*wn/16;
  return(answer);
}

/*******************************************************************************/
/*                              CONVOLUTION ROUTINES                           */
/*******************************************************************************/
/* Returns a single sample located at (i, j) for a given color (red, green, or
   blue pixels) for the convolution of image with w_dot.  */
double sample_w_g_dot(image * pic, int i, int j, int color)
{
  int m, n, index, tx, ty;
  double tsum = 0.0;
  for(m = -1; m < 2; m++)  {
    for(n = -1; n < 2; n++)  {
      tx = i + m;
      if(tx < 0) tx = -tx;
      if(tx >= pic->xsize)   tx = 2*pic->xsize - tx - 1;
      ty = j + n;
      if(ty < 0) ty = -ty;
      if(ty >= pic->ysize)   ty = 2*pic->ysize - ty - 1;
      index = ty*pic->xsize + tx;
      tsum += w_g_dot(m,n) * pic->pixel[index][color];
    }
  }
  return(tsum);
}

/* Returns a single sample located at (i, j) for a given color (red, green, or
   blue pixels) for the convolution of image with w the gaussian kernel.  */
double sample_w_g(image * pic, int i, int j, int color)
{
  int m, n, index, tx, ty;
  double tsum = 0.0;
  for(m = -2; m < 3; m++)  {
    for(n = -2; n < 3; n++)  {
      tx = i + m;
      if(tx < 0) tx = -tx;
      if(tx >= pic->xsize)   tx = 2*pic->xsize - tx - 1;

      ty = j + n;
      if(ty < 0) ty = -ty;
      if(ty >= pic->ysize)   ty = 2*pic->ysize - ty - 1;

      index = ty*pic->xsize + tx;
      tsum += w_g(m,n) * pic->pixel[index][color];
    }
  }
  return(tsum);
}

/* Returns a single sample located at (i, j) for a given color (red,
   green, or blue pixels) for the convolution of image with oriented
   derivative filter d1.  */
double sample_d1(image * pic, int i, int j, int color, double mult)
{
  int index;

  if(i >= 0 && i <= pic->xsize)  {
    index = j*pic->xsize + i;
    if(i == 0) return(pic->pixel[index][color])*mult/3.5;
    else if(i == pic->xsize)  return(-(pic->pixel[index-1][color])*mult/3.5);
    else  return((pic->pixel[index][color] - pic->pixel[index-1][color])*mult);
  }
  else
    return(-111000);
}

/* Returns a single sample located at (i, j) for a given color (red,
   green, or blue pixels) for the convolution of image with oriented
   derivative filter d2.  */
double sample_d2(image * pic, int i, int j, int color, double mult)
{
  double a, b;

  if(i >= 0 && i <= pic->xsize)  {
    if(i > 0 && i <= pic->xsize && j < pic->ysize)  {
      a = pic->pixel[j*pic->xsize + i - 1][color];
    }
    else a = 0;
    if(j > 0 && j <= pic->ysize && i < pic->xsize)  {
      b = pic->pixel[(j-1)*pic->xsize + i][color];
    }
    else b = 0;
    if(a == 0 || b == 0)
      return((b - a)*sqrt2inv*mult/3.5);
    else
      return((b - a)*sqrt2inv*mult);
  }
  else
    return(-111000);
}

/* Returns a single sample located at (i, j) for a given color (red,
   green, or blue pixels) for the convolution of image with oriented
   derivative filter d3.  */
double sample_d3(image * pic, int i, int j, int color, double mult)
{
  double a, b;

  if(j >= 0 && j <= pic->ysize)  {
    if(j < pic->ysize)
      a = pic->pixel[j*pic->xsize + i][color];
    else a = 0;
    if(j > 0)
      b = pic->pixel[(j-1)*pic->xsize + i][color];
    else b = 0;
    if(a == 0 || b == 0)
      return((b - a)*mult/3.5);
    else
      return((b - a)*mult);
  }
  else
    return(-111000);
}

/* Returns a single sample located at (i, j) for a given color (red,
   green, or blue pixels) for the convolution of image with oriented
   derivative filter d4.  */
double sample_d4(image * pic, int i, int j, int color, double mult)
{
  double a, b;

  if(j >= 0 && j <= pic->xsize)  {
    if(j < pic->ysize && i < pic->xsize)  {
      a = pic->pixel[j*pic->xsize + i][color];
    }
    else a = 0;
    if(j > 0 && i > 0)  {
      b = pic->pixel[(j-1)*pic->xsize + (i - 1)][color];
    }
    else b = 0;
    if(a == 0 || b == 0)
      return((b - a)*sqrt2inv*mult/3.5);
    else
      return((b - a)*sqrt2inv*mult);
  }
  else
    return(-111000);
}

/* Convolves the image g with the Gaussian kernel w */
image * conv_w_g_image(image *g)
{
  image *newpic;
  int xnew, ynew, x, y, index;
  newpic = (image *)malloc(sizeof(image));
  xnew = g->xsize;
  ynew = g->ysize;
  newpic->pixel = (Pixel *)malloc(xnew*ynew*sizeof(Pixel));
  for(x = 0; x < xnew; x++)  {
    for(y = 0; y < ynew; y++)  {
      index = y*xnew + x;
      newpic->pixel[index][0] = sample_w_g(g, x, y, 0);
      newpic->pixel[index][1] = sample_w_g(g, x, y, 1);
      newpic->pixel[index][2] = sample_w_g(g, x, y, 2);
    }
  }
  newpic->xsize = xnew;
  newpic->ysize = ynew;
  return(newpic);
}

/* Convolves the image g with the seperated Gaussian kernel w_dot */
image * conv_w_g_dot_image(image *g)
{
  image *newpic;
  int xnew, ynew, x, y, index;
  newpic = (image *)malloc(sizeof(image));
  xnew = g->xsize;
  ynew = g->ysize;
  newpic->pixel = (Pixel *)malloc(xnew*ynew*sizeof(Pixel));
  for(y = 0; y < ynew; y++)  {
    for(x = 0; x < xnew; x++)  {
      index = y*xnew + x;
      newpic->pixel[index][0] = sample_w_g_dot(g, x, y, 0);
      newpic->pixel[index][1] = sample_w_g_dot(g, x, y, 1);
      newpic->pixel[index][2] = sample_w_g_dot(g, x, y, 2);
    }
  }
  newpic->xsize = xnew;
  newpic->ysize = ynew;
  return(newpic);
}

/* Convolves the image g with the first derivative filter d1 timesed
   by a multiplier mult */
image * conv_d1_image(image *g, double mult)
{
  image *newpic;
  int x, y, index;
  newpic = (image *)malloc(sizeof(image));
  newpic->xsize = g->xsize+1;
  newpic->ysize = g->ysize;
  newpic->pixel = (Pixel *)malloc(newpic->xsize*newpic->ysize*sizeof(Pixel));
  for(y = 0; y < newpic->ysize; y++)  {
    for(x = 0; x < newpic->xsize; x++)  {
      index = y*newpic->xsize + x;
      newpic->pixel[index][0] = sample_d1(g, x, y, 0, mult);
      newpic->pixel[index][1] = sample_d1(g, x, y, 1, mult);
      newpic->pixel[index][2] = sample_d1(g, x, y, 2, mult);
    }
  }
  return(newpic);
}

/* Convolves the image g with the first derivative filter d2 timesed
   by a multiplier mult */
image * conv_d2_image(image *g, double mult)
{
  image *newpic;
  int x, y, index;
  newpic = (image *)malloc(sizeof(image));
  newpic->xsize = g->xsize+1;
  newpic->ysize = g->ysize+1;
  newpic->pixel = (Pixel *)malloc(newpic->xsize*newpic->ysize*sizeof(Pixel));
  for(y = 0; y < newpic->ysize; y++)  {
    for(x = 0; x < newpic->xsize; x++)  {
      index = y*newpic->xsize + x;
      newpic->pixel[index][0] = sample_d2(g, x, y, 0, mult);
      newpic->pixel[index][1] = sample_d2(g, x, y, 1, mult);
      newpic->pixel[index][2] = sample_d2(g, x, y, 2, mult);
    }
  }
  return(newpic);
}
```

```
/* Convolves the image g with the first derivative filter d3 timesed
   by a multiplier mult */
image * conv_d3_image(image *g, double mult)
{
  image *newpic;
  int x, y, index;
  newpic = (image *)malloc(sizeof(image));
  newpic->xsize = g->xsize;
  newpic->ysize = g->ysize+1;
  newpic->pixel = (Pixel *)malloc(newpic->xsize*newpic->ysize*sizeof(Pixel));
  for(y = 0; y < newpic->ysize; y++)  {
    for(x = 0; x < newpic->xsize; x++)  {
      index = y*newpic->xsize + x;
      newpic->pixel[index][0] = sample_d3(g, x, y, 0, mult);
      newpic->pixel[index][1] = sample_d3(g, x, y, 1, mult);
      newpic->pixel[index][2] = sample_d3(g, x, y, 2, mult);
    }
  }
  return(newpic);
}

/* Convolves the image g with the first derivative filter d4 timesed
   by a multiplier mult */
image * conv_d4_image(image *g, double mult)
{
  image *newpic;
  int x, y, index;
  newpic = (image *)malloc(sizeof(image));
  newpic->xsize = g->xsize+1;
  newpic->ysize = g->ysize+1;
  newpic->pixel = (Pixel *)malloc(newpic->xsize*newpic->ysize*sizeof(Pixel));
  for(y = 0; y < newpic->ysize; y++)  {
    for(x = 0; x < newpic->xsize; x++)  {
      index = y*newpic->xsize + x;
      newpic->pixel[index][0] = sample_d4(g, x, y, 0, mult);
      newpic->pixel[index][1] = sample_d4(g, x, y, 1, mult);
      newpic->pixel[index][2] = sample_d4(g, x, y, 2, mult);
    }
  }
  return(newpic);
}

/*******************************************************************************/
/*                            RESIZING ROUTINES                                */
/*******************************************************************************/
/* Returns a single sample located at (i, j) for the REDUCED image of pic
   color is the indicator for red, green, or blue pixels.  */
double reducepixel_g(image * pic, int i, int j, int color)
{
  int m, n, index, tx, ty;
  double tsum = 0.0;
  for(m = -2; m < 3; m++)  {
    for(n = -2; n < 3; n++)  {
      tx = 2*i + m;
      if(tx < 0 || tx >= pic->xsize)   continue;

      ty = 2*j + n;
      if(ty < 0 || ty >= pic->ysize)   continue;

      index = ty*pic->xsize + tx;
      tsum += w_g(m,n) * pic->pixel[index][color];
    }
  }
  return(tsum);
}

/* Filters and subsamples the image g to get the next level in the
   gradient pyramid  */
```

```
image * reduce_g(image *g)
{
  image *newpic;
  int xnew, ynew, x, y, index;
  newpic = (image *)malloc(sizeof(image));
  xnew = (g->xsize - 1)/2 + 1;
  ynew = (g->ysize - 1)/2 + 1;
  newpic->pixel = (Pixel *)malloc(xnew*ynew*sizeof(Pixel));
  for(x = 0; x < xnew; x++)  {
    for(y = 0; y < ynew; y++)  {
      index = y*xnew + x;
      newpic->pixel[index][0] = (int)reducepixel_g(g, x, y, 0);
      newpic->pixel[index][1] = (int)reducepixel_g(g, x, y, 1);
      newpic->pixel[index][2] = (int)reducepixel_g(g, x, y, 2);
    }
  }
  newpic->xsize = xnew;
  newpic->ysize = ynew;
  return(newpic);
}

/* Returns a single sample located at (i, j) for the EXPANDED image of pic
   color is the indicator for red, green, or blue pixels.  */
double expandpixel_g(image * pic, int i, int j, int color)
{
  int m, n, index, tx, ty;
  double tsum = 0.0;
  for(m = -2; m < 3; m++)  {
    for(n = -2; n < 3; n++)  {
      if( ((i-m)%2 == 0) && ((j-n)%2 == 0) )  {
        tx = (i-m)/2;
        if(tx < 0 || tx >= pic->xsize)   continue;

        ty = (j - n)/2;
        if(ty < 0 || ty >= pic->ysize)   continue;

        index = ty*pic->xsize + tx;
        tsum += w_g(m,n) * pic->pixel[index][color];
      }
    }
  }
  return(4*tsum);
}

/* Upsamples and interpolates the image g to get a lower resolution image
   of the previoius level in the pyramid*/
image * expand_g(image *g)
{
  image *newpic;
  int xnew, ynew, x, y, index;
  newpic = (image *)malloc(sizeof(image));
  xnew = (g->xsize - 1)*2 + 1;
  ynew = (g->ysize - 1)*2 + 1;
  newpic->pixel = (Pixel *)malloc(xnew*ynew*sizeof(Pixel));
  for(x = 0; x < xnew; x++)  {
    for(y = 0; y < ynew; y++)  {
      index = y*xnew + x;
      newpic->pixel[index][0] = (int)expandpixel_g(g, x, y, 0);
      newpic->pixel[index][1] = (int)expandpixel_g(g, x, y, 1);
      newpic->pixel[index][2] = (int)expandpixel_g(g, x, y, 2);
    }
  }
  newpic->xsize = xnew;
  newpic->ysize = ynew;
  return(newpic);
}

/*  Adds the images of the four different gradient pyramids associated
    with d1, d2, d3, and d4.  It returns the FSD leplacian pyramid. */
```

```
image *add11to14(image *l1, image *l2, image *l3, image *l4)
{
  image *newpic;
  int xnew, ynew, x, y, index;
  int xtemp, ytemp;
  xnew = l3->xsize;
  ynew = l1->ysize;
  newpic = (image *)malloc(sizeof(image));
  newpic->pixel = (Pixel *)malloc(xnew*ynew*sizeof(Pixel));
  xtemp = xnew + 2;
  for(y = 0; y < ynew; y++)  {
    for(x = 0; x < xnew; x++)  {
      index = y*xnew + x;
      ytemp = y + 1;
      newpic->pixel[index][0] = l1->pixel[y    *xtemp  + x + 1][0] +
                                l2->pixel[ytemp*xtemp  + x + 1][0] +
                                l3->pixel[ytemp*(xnew) + x    ][0] +
                                l4->pixel[ytemp*xtemp  + x + 1][0];
      newpic->pixel[index][1] = l1->pixel[y    *xtemp  + x + 1][1] +
                                l2->pixel[ytemp*xtemp  + x + 1][1] +
                                l3->pixel[ytemp*(xnew) + x    ][1] +
                                l4->pixel[ytemp*xtemp  + x + 1][1];
      newpic->pixel[index][2] = l1->pixel[y    *xtemp  + x + 1][2] +
                                l2->pixel[ytemp*xtemp  + x + 1][2] +
                                l3->pixel[ytemp*(xnew) + x    ][2] +
                                l4->pixel[ytemp*xtemp  + x + 1][2];
    }
  }
  newpic->xsize = xnew;
  newpic->ysize = ynew;
  return(newpic);
}

/*********************************************************************************/
/*                              FSD Conversions                                  */
/*********************************************************************************/
/* Converts the d1 gradient image into the d1 FSD Leplacian image.  */
image *gen_gradient_FSD_d1(image *d)
{
  image * temp;
  int x, y, index;

  temp = (image *)malloc(sizeof(image));
  temp->xsize = d->xsize+1;
  temp->ysize = d->ysize;
  temp->pixel = (Pixel *)malloc(temp->xsize*temp->ysize*sizeof(Pixel));
  for(x = 0; x < temp->xsize; x++)  {
    for(y = 0; y < temp->ysize; y++)  {
      index = y*temp->xsize + x;
      temp->pixel[index][0] = sample_d1(d, x, y, 0, -eighth);
      temp->pixel[index][1] = sample_d1(d, x, y, 1, -eighth);
      temp->pixel[index][2] = sample_d1(d, x, y, 2, -eighth);
    }
  }
  return(temp);
}

/* Converts the d2 gradient image into the d2 FSD Leplacian image.  */
image *gen_gradient_FSD_d2(image *g)
{
  image * temp;
  int x, y, index;

  temp = (image *)malloc(sizeof(image));
  temp->xsize = g->xsize + 1;
  temp->ysize = g->ysize + 1;
  temp->pixel = (Pixel *)malloc(temp->xsize*temp->ysize*sizeof(Pixel));
  for(x = 0; x < temp->xsize; x++)  {
    for(y = 0; y < temp->ysize; y++)  {
        index = y*temp->xsize + x;
        temp->pixel[index][0] = sample_d2(g, x, y, 0, -eighth);
        temp->pixel[index][1] = sample_d2(g, x, y, 1, -eighth);
        temp->pixel[index][2] = sample_d2(g, x, y, 2, -eighth);
      }
    }
  return(temp);
}

/* Converts the d3 gradient image into the d3 FSD Leplacian image.  */
image *gen_gradient_FSD_d3(image *g)
{
  image * temp;
  int x, y, index;

  temp = (image *)malloc(sizeof(image));
  temp->xsize = g->xsize;
  temp->ysize = g->ysize + 1;
  temp->pixel = (Pixel *)malloc(temp->xsize*temp->ysize*sizeof(Pixel));
  for(x = 0; x < temp->xsize; x++)  {
    for(y = 0; y < temp->ysize; y++)  {
      index = y*temp->xsize + x;
      temp->pixel[index][0] = sample_d3(g, x, y, 0, -eighth);
      temp->pixel[index][1] = sample_d3(g, x, y, 1, -eighth);
      temp->pixel[index][2] = sample_d3(g, x, y, 2, -eighth);
    }
  }
  return(temp);
}

/* Converts the d4 gradient image into the d4 FSD Leplacian image.  */
image *gen_gradient_FSD_d4(image *g)
{
  image * temp;
  int x, y, index;

  temp = (image *)malloc(sizeof(image));
  temp->xsize = g->xsize + 1;
  temp->ysize = g->ysize + 1;
  temp->pixel = (Pixel *)malloc(temp->xsize*temp->ysize*sizeof(Pixel));
  for(x = 0; x < temp->xsize; x++)  {
    for(y = 0; y < temp->ysize; y++)  {
      index = y*temp->xsize + x;
      temp->pixel[index][0] = sample_d4(g, x, y, 0, -eighth);
      temp->pixel[index][1] = sample_d4(g, x, y, 1, -eighth);
      temp->pixel[index][2] = sample_d4(g, x, y, 2, -eighth);
    }
  }
  return(temp);
}

/* Converts the FSD Leplacian image into a Leplacian image.  */
image * FSD_to_RE(image *g)
{
  image *temp, *t2;

  t2 = conv_w_g_image(g);
  temp = add_sign(g, t2);
  free_image(t2);

  return(temp);
}

#endif
```

```
/*********************************** files.cpp ***********************************/
/*   This file contains the code for reading an image file, and storing the     */
/*   pixel information into an image structure.  The loadfile routine is set    */
/*   for portable bitmaps, but can be implemented for any image storage type.  */
/*   Image memory routines are also set up, as well as routines to resize the  */
/*   image and copy the image.  (used for portable bitmaps only)                */
/********************************************************************************/
#ifndef FILES_CPP
#define FILES_CPP

#include "thesis.h"

/* This is the type of a pixel 0 is red, 1 is green, and index 2 is blue */
typedef double    Pixel[3];

/* This is the structure to store an image as a two dimentional array of pixels */
typedef struct image {
  int xsize, ysize;
  Pixel *pixel;
} image;

/* Functions which are defined later */
void free_image(image * i1);
void initfile(FILE **infile, char * s, int *xs, int *ys);
void loadfile(image *filestruct, char * name);
void interpolate_pixel_nn(image * g, int xnew, int ynew, int x, int y, DPoint color);
void interpolate_pixel_bi(image * g, int xnew, int ynew, int x, int y, DPoint color);
image * resize(image *i1, int n);
image * copy_image(image *i);

/* Function definitions */
/* this function copies the image passed in to another image */
/* basically it allocates memory and copies the contents of the image i and
   returns a pointer to the new image */
image * copy_image(image *i)
{
  int index, x, y;
  image *d;
  d = (image *)malloc(sizeof(image));
  d->pixel = (Pixel *)malloc(i->xsize*i->ysize*sizeof(Pixel));

  d->xsize = i->xsize;
  d->ysize = i->ysize;

  for(y = 0; y < i->ysize; y++)  {
    for(x = 0; x < i->xsize; x++)  {
      index = y*i->xsize + x;
      d->pixel[index][0] = (double)i->pixel[index][0];
      d->pixel[index][1] = (double)i->pixel[index][1];
      d->pixel[index][2] = (double)i->pixel[index][2];
    }
  }
  return(d);
}

/* This frees the memory used to store the image */
void free_image(image * i1)
{
  if(i1 != NULL)  {
    if(i1->pixel != NULL)  {
      free(i1->pixel);
    }
    free(i1);
  }
  i1 = NULL;
  return;
}

/* reads the header of a ppm file and returns the x and y dimentions in xs and ys */
```

```
void initfile(FILE **infile, char * s, int *xs, int *ys)
{
       unsigned char dummystring[256];
       unsigned char final;
  int xsize, ysize, dummy;

  (*infile) = fopen(s, "r");
  if (!(*infile))
              return;

       fscanf((*infile), "%s",dummystring);
  fscanf((*infile), "%c",&final);
       while(final != '\n')
    fscanf((*infile), "%c",&final);
  fscanf((*infile), "%c",&final);
       while(final != '\n')
    fscanf((*infile), "%c",&final);
  fscanf((*infile), "%d%d%d%c",&xsize, &ysize, &dummy, &final);
       while(final != '\n')
    fscanf((*infile), "%c",&final);

  (*xs) = xsize;
  (*ys) = ysize;
}

/* this function loads a file into the image pointer filestruct
   name is a string which is the name of the file to be read in.
   the image must be saved as a ppm to be read using this function */
void loadfile(image *filestruct, char * name)
{
  int xsize, ysize, x, y, index;
  FILE * infile;
  int red, green, blue;
  initfile(&infile, name, &xsize, &ysize);
  filestruct->pixel = (Pixel *)malloc(xsize*ysize*sizeof(Pixel));
  for(y = 0; y < ysize; y++)  {
    for(x = 0; x < xsize; x++)  {
      fscanf(infile, "%d%d%d", &red, &green, &blue);
      index = y*xsize + x;
      filestruct->pixel[index][0] = red;
      filestruct->pixel[index][1] = green;
      filestruct->pixel[index][2] = blue;
    }
  }
  filestruct->xsize = xsize;
  filestruct->ysize = ysize;
  fclose(infile);
}

/* This function copies the contents of the image to a ppm file with the name of
   the string passed in */
void savefile(image *filestruct, char * name)
{
  int numcol = 255, x, y, xsize, ysize, index;
  FILE * outfile;
  outfile = fopen(name,"w");
  if (outfile == NULL)
              return;
  xsize = filestruct->xsize;
  ysize = filestruct->ysize;
  fprintf(outfile, "P3\n# Created by Ted Meek\n");
  fprintf(outfile, "%d %d %d\n", xsize, ysize, numcol);
  for(y = 0; y < ysize; y++)  {
    for(x = 0; x < xsize; x++)  {
      index = y*xsize + x;
        fprintf(outfile, "%d %d %d ", (int)abs(filestruct->pixel[index][0]),
                                      (int)abs(filestruct->pixel[index][1]),
                                      (int)abs(filestruct->pixel[index][2]));
    }
  }
  fclose(outfile);
}

/* This function copies the contents of the image to a ppm file with the name of
   the string passed in.  It is a grayscale of either the red, blue, or green image.
   To save red, color = 0, green color = 1, and blue color = 2*/
void savefile_onecolor(image *filestruct, char * name, int color)
{
  int numcol = 255, x, y, xsize, ysize, index;
  int red, green, blue;
  FILE * outfile;
  outfile = fopen(name,"w");
  if (outfile == NULL)
             return;
  xsize = filestruct->xsize;
  ysize = filestruct->ysize;
  fprintf(outfile, "P3\n# Created by Ted Meek\n");
  fprintf(outfile, "%d %d %d\n", xsize, ysize, numcol);
  for(y = 0; y < ysize; y++)  {
    for(x = 0; x < xsize; x++)  {
      index = y*xsize + x;
      red = (int)filestruct->pixel[index][color];
      green = (int)filestruct->pixel[index][color];
      blue = (int)filestruct->pixel[index][color];
      fprintf(outfile, "%d %d %d ", red, green, blue);
    }
  }
  fclose(outfile);
}


/* This resizes the image to the closest, next-smallest power of 2 dimention.
   If n = 5, it will resize to 2^5, otherwise it will use 2^4.
   This is necessary for creating a pyramid to represent the image which is
   at least 4 levels deep.  Bilinear interpolation is used.  */
image * resize(image *i1, int n)
{
  image *newpic;
  int xnew, ynew, preMc, preMr, index, N, x, y;
  DPoint color;
  if(n == 5) N = 32; else N = 16;

  preMc = (int)floor((double)i1->xsize/N);
  preMr = (int)floor((double)i1->ysize/N);
  xnew = preMc*N + 1;
  ynew = preMr*N + 1;

  newpic = (image *)malloc(sizeof(image));
  newpic->pixel = (Pixel *)malloc(xnew*ynew*sizeof(Pixel));
  newpic->xsize = xnew;
  newpic->ysize = ynew;

  for(y = 0; y < ynew; y++)  {
    for(x = 0; x < xnew; x++)  {
      index = y*xnew + x;
      interpolate_pixel_bi(i1, xnew, ynew, x, y, color);
      newpic->pixel[index][0] = color[0];
      newpic->pixel[index][1] = color[1];
      newpic->pixel[index][2] = color[2];
    }
  }
  return(newpic);
}

/* This resizes the image to the closest, next-smallest power of 2 dimention.
   If n = 5, it will resize to 2^5, otherwise it will use 2^4.
   This is necessary for creating a pyramid to represent the image which is
   at least 4 levels deep.  Nearest Neighbor interpolation is used.  */
```

```
image * resize_nn(image *i1, int n)
{
  image *newpic;
  int xnew, ynew, preMc, preMr, index, N, x, y;
  DPoint color;
  if(n == 5) N = 32; else N = 16;

  preMc = (int)floor((double)i1->xsize/N);
  preMr = (int)floor((double)i1->ysize/N);
  xnew = preMc*N + 1;
  ynew = preMr*N + 1;

  newpic = (image *)malloc(sizeof(image));
  newpic->pixel = (Pixel *)malloc(xnew*ynew*sizeof(Pixel));
  newpic->xsize = xnew;
  newpic->ysize = ynew;

  for(y = 0; y < ynew; y++)  {
    for(x = 0; x < xnew; x++)  {
      index = y*xnew + x;
      interpolate_pixel_nn(i1, xnew, ynew, x, y, color);
      newpic->pixel[index][0] = color[0];
      newpic->pixel[index][1] = color[1];
      newpic->pixel[index][2] = color[2];
    }
  }
  return(newpic);
}

/* This resizes the image to a defined x and y size; defined by xnew
   and ynew.  Bilinear interpolation is used.  */
void resize_define(image *i1, int xnew, int ynew)
{
  Pixel *pic;
  int index, x, y;
  DPoint color;

  pic = (Pixel *)malloc(xnew*ynew*sizeof(Pixel));

  for(y = 0; y < ynew; y++)  {
    for(x = 0; x < xnew; x++)  {
      index = y*xnew + x;
      interpolate_pixel_bi(i1, xnew, ynew, x, y, color);
      pic[index][0] = color[0];
      pic[index][1] = color[1];
      pic[index][2] = color[2];
    }
  }
  free(i1->pixel);
  i1->pixel = pic;
  i1->xsize = xnew;
  i1->ysize = ynew;
  return;
}

/*Nearest neighbor interpolation */
void interpolate_pixel_nn(image * g, int xnew, int ynew, int x, int y, DPoint color)
{
  int fx, fy, index;
  double xval, yval;

  xval = (double)(x*g->xsize)/(double)xnew;
  yval = (double)(y*g->ysize)/(double)ynew;

  fx = (int)round(xval);
  fy = (int)round(yval);
  index = fy*g->xsize + fx;

  color[0] = g->pixel[index][0];
  color[1] = g->pixel[index][1];
  color[2] = g->pixel[index][2];
  return;
}

/*bilinear interpolation */
void interpolate_pixel_bi(image * g, int xnew, int ynew, int x, int y, DPoint color)
{
  double i1, i2, i3, i4;
  int fx, fy, cx, cy;
  double xval, yval, s1, s2, s3, s4, error = .0000000001;

  xval = (x*g->xsize)/xnew;
  yval = (y*g->ysize)/ynew;

  fx = (int)floor(xval+error);
  cx = (int)ceil(xval);
  if(fx == cx && cx != (g->xsize -1)) cx++;
  fy = (int)floor(yval+error);
  cy = (int)ceil(yval);
  if(fy == cy && cy != (g->ysize -1)) cy++;

  s1 = (xval - fx)/2;
  s2 = 0.5 - s1;
  s3 = (yval - fy)/2;
  s4 = 0.5 - s3;

  i1 = g->pixel[fy*g->xsize + fx][0];
  i2 = g->pixel[fy*g->xsize + cx][0];
  i3 = g->pixel[cy*g->xsize + fx][0];
  i4 = g->pixel[cy*g->xsize + cx][0];
  color[0] = (s1*i1 + s2*i2 + s3*i3 + s4*i4);
  i1 = g->pixel[fy*g->xsize + fx][1];
  i2 = g->pixel[fy*g->xsize + cx][1];
  i3 = g->pixel[cy*g->xsize + fx][1];
  i4 = g->pixel[cy*g->xsize + cx][1];
  color[1] = (s1*i1 + s2*i2 + s3*i3 + s4*i4);

  i1 = g->pixel[fy*g->xsize + fx][2];
  i2 = g->pixel[fy*g->xsize + cx][2];
  i3 = g->pixel[cy*g->xsize + fx][2];
  i4 = g->pixel[cy*g->xsize + cx][2];
  color[2] = (s1*i1 + s2*i2 + s3*i3 + s4*i4);
  return;
}

#endif
```